SYMMETRIC MULTIPROCESSING VIRTUALIZATION

by

Gabriel Southern

A Thesis
Submitted to the
Graduate Faculty
of
George Mason University
In Partial fulfillment of
The Requirements for the Degree
of
Master of Science
Computer Engineering

Committee:

[illegible]	Dr. David Hwang, Thesis Director
KGaj	Dr. Kris Gaj, Committee Member
[illegible]	Dr. Hakan Aydin, Committee Member
[illegible]	Dr. Andre Manitius, Department Chair
[illegible]	Dr. Lloyd J. Griffiths, Dean, The Volgenau School of Information Technology and Engineering

Date: July 28, 2008

Summer Semester 2008
George Mason University
Fairfax, VA

Symmetric Multiprocessing Virtualization

A thesis submitted in partial fulfillment of the requirements for the degree of Master of Science at George Mason University

By

Gabriel Southern
Bachelor of Science
University of Virginia, 2002

Director: Dr. David Hwang, Professor
Department of Electrical and Computer Engineering

Summer Semester 2008
George Mason University
Fairfax, VA

Acknowledgments

I would like to thank all of the faculty, staff, and administrators at George Mason University who helped me as a student and made it possible for me to complete this thesis. There are three individuals I would like to acknowledge for special thanks:

Dr. Ronald Barnes, whose expertise in the field of computer architecture was vital for my work on this thesis. His instruction in the classroom helped me find the field of computer engineering I find most interesting, and his direction as an adviser has helped me to learn and understand the field of virtualization. I am especially grateful to him for agreeing to continue working with me even after he left George Mason to pursue opportunities in his home state at the University of Oklahoma.

Dr. Kris Gaj, whose tireless work as an instructor helped me advance beyond simply learning existing material to doing independent research on new ideas. I am especially grateful for the opportunities he provided to extend into research papers the projects I worked on in his classes. This experience was extremely beneficial to me as I was working on this thesis.

Dr. David Hwang, who graciously agreed to serve as my adviser after Dr. Barnes departed for the University of Oklahoma. His guidance was critical to keeping my research on track, and he made sure that I had the resources necessary to conduct my research. Without his assistance it would not have been possible for me to complete this research.

Finally I would like to thank my parents for their assistance throughout my education, including my work on this thesis.

Table of Contents

Page

List of Tables vi

List of Figures vii

Abstract viii

1 Introduction 1

2 Virtualization Overview 3

2.1 System Virtualization 4

2.2 Virtualization Marketplace 6

2.2.1 Workstation Virtualization 6

2.2.2 Server Virtualization 7

2.3 Virtualization History 9

2.4 Virtualization Requirements 10

2.5 VMware Virtualization 12

2.5.1 CPU Virtualization 13

2.5.2 Memory Virtualization 15

2.5.3 I/O Virtualization 17

2.6 Xen 18

2.6.1 CPU Virtualization 19

2.6.2 Memory Virtualization 19

2.6.3 I/O Virtualization 20

2.7 Hardware Virtualization Extensions 21

2.7.1 VT-x overview 21

2.7.2 VT-x Implementation in Xen 22

3 Symmetric Multiprocessor Virtualization 24

3.1 Computer Architecture History and Trends 24

3.2 Multiprocessing 26

3.2.1 Symmetric Multiprocessing 27

3.2.2 SMP Workloads 27

3.3 SMP Virtual Machine Challenges 29

3.3.1 Spinlocks 29

3.3.2 Co-scheduling 30
3.4 Proposed optimizations for SMP VM scheduling 31
3.4.1 Avoiding Lock-Holder Preemption 31
3.4.2 Hardware Support for Spin Management 32
4 SMP Virtualization in ESX and Xen 34
4.1 CPU scheduling background 34
4.1.1 ESX scheduling 34
4.1.2 Xen CPU Scheduling 37
4.2 Experimental Setup 40
4.2.1 System Configuration 40
4.2.2 Benchmarks 41
4.3 ESX Benchmark Results 43
4.3.1 Single Threaded Benchmarks 43
4.3.2 Multithreaded Benchmarks 45
4.4 Xen Benchmark Results 47
4.4.1 Single Threaded Equal VCPU Allocation Results 47
4.4.2 Single Threaded Different VCPU Allocation Results 50
4.4.3 Multithreaded benchmarks 53
4.5 ESX vs. Xen Summary 56
5 Dynamic VCPU Allocation 58
5.1 Proposed Optimization Background 60
5.2 Dynamic VCPU allocation 60
5.2.1 Communicate current guest OS parallelism to VMM 61
5.2.2 Communicate optimal parallelism value from VMM to guest 61
5.2.3 Proposed Algorithm 63
5.3 Summary 64
6 Conclusion and Future Work 65
A VCPU Allocation Source Code 68
Bibliography 72

List of Tables

Table — Page

4.1 Test system hardware configuration 41

4.2 VMware ESX software configuration 41

4.3 Xen software configuration 41

4.4 ESX and Xen comparison 56

List of Figures

Figure Page

2.1 System virtualization example 10
2.2 Binary translation control flow 14
2.3 Shadow page table operation 16
2.4 IA-32 privilege rings 20
2.5 Intel VT-x operations 22
4.1 ESX scheduling example 36
4.2 Xen Scheduling Example Initial State 38
4.3 Xen Scheduling Example PCPU P1 39
4.4 Xen Scheduling Example PCPU P2 40
4.5 ESX: 1 VM with 1 instance of 445.gobmk 44
4.6 ESX: 8 VMs with 1 instance of 445.gobmk 44
4.7 ESX: 1 VM executing 332.ammp_m 46
4.8 ESX: 8 VMs executing 332.ammp_m 46
4.9 Xen: 8 VMs with 1 instance of 445.gobmk 47
4.10 Xen: 8 VMs executing 8 instances of 445.gobmk 49
4.11 Xen: Different numbers of VCPUs per VM 51
4.12 Xen: Different priority per VM 52
4.13 Xen: Percentage CPU allocation 53
4.14 Xen: Single VM executing 332.ammp_m 54
4.15 Xen: Eight VMs executing 332.ammp_m 55
5.1 Speedup for 332.ammp_m: Optimal vs. Actual 59

Abstract

SYMMETRIC MULTIPROCESSING VIRTUALIZATION

Gabriel Southern, MS

George Mason University, 2008

Thesis Director: Dr. David Hwang

The reemergence of system virtualization, and the introduction of the single-chip multiprocessor, are two technologies that are altering the landscape of modern general-purpose computing architecture. System virtualization is a technology that uses software to partition the resources of a single physical computer into multiple virtual machines (VMs). Multiprocessing combines the capabilities of multiple CPUs into a single computer system. Virtualization continues to play an important role in efficiently utilizing multiprocessor systems; however, developing efficient multiprocessor virtual machines is a significant challenge.

Multicore processors — those that combine multiple logical CPUs in a single physical socket — now dominate the general purpose microprocessor marketplace. These processors require operating system support for multiprocessing, and the most common form of multiprocessing is symmetric multiprocessing (SMP), where all processors have identical behavior and any processor can perform any task.

System virtualization introduces additional complexities for implementing SMP because VMs have different timing behavior from physical computers. In this thesis I examine two approaches to implementing SMP in VMs used by two leading virtualization platforms: VMware ESX server and the Xen hypervisor.

I demonstrate the comparative advantages of these two different virtualization systems, and propose a new optimization technique to improve performance through dynamic virtual CPU (VCPU) allocation.

I find that Xen performs well when executing independent single threaded applications that do not require any synchronization between the processes running on multiple processors. However, Xen scales poorly for applications that require synchronization between multiple threads.

ESX uses a more conservative scheduling algorithm, called co-scheduling, that provides more balanced performance for both single threaded and multithreaded workloads. However, ESX has some significant optimization that provide improved performance over a naive implementation of co-scheduling.

Finally, I propose an algorithm for dynamic allocation of VCPU resources that can be used by Xen to improve the efficiency of multithreaded workloads. I provide a simulation for this algorithm and discuss the tasks required for a complete implementation of the algorithm.

Chapter 1: Introduction

System virtualization technology uses software to partition the resources of a single physical computer system into multiple virtual machines. A virtual machine (VM) is simply a replica of a physical machine and its behavior is unchanged from the physical system, outside of possible differences in timing characteristics for some operations [1]. System virtualization has a long history in the world of mainframe computers, but only recently has it become popular in general purpose computing [2].

The x86 architecture is the most popular instruction set architecture (ISA) in the general purpose computing marketplace. When the architecture was first developed it did not provide support for virtualization, and indeed for some time it was thought that virtualization was not possible using the x86 architecture [3]. However, as the x86-based computer became more powerful, researchers at Stanford University discovered a way to virtualize the x86 architecture using software that borrowed heavily from ideas in emulation technology. These researchers founded a company named VMware, which today is the leading developer of virtualization software for the x86 architecture. Their most advanced high-performance virtualization product is VMware ESX [4].

After VMware was founded, researchers at the University of Cambridge proposed an alternative x86 virtualization technique called paravirtualization [5]. The open-source software implementation of their idea was named Xen. Their implementation was closely affiliated with the Linux operating system and they chose to offer it as an open source product similar to Linux. Xen has since grown in popularity and is supported as a stand-alone distribution by several different vendors, is also incorporated as part of several commercial Linux distributions, and is included with Sun OpenSolaris.

Both Xen and ESX can use multiprocessing to provide increased performance for VMs by taking advantage of multiple CPUs operating in a symmetric multiprocessing (SMP) system. They both also offer the ability to assign multiple virtual CPUs to a single virtual machine; however, in this case the performance benefits are not necessarily as clear. Using SMP VMs can reduce the overall system performance, depending on the types of workload running inside the VMs and the overall system configuration. Both VMware [6,7] and Xen [8] recommend that system administrators carefully analyze their system workload before deciding to configure SMP VMs. While the vendors for these two different virtualization software solutions offer the same advice about configuring SMP VM, they have very different approaches for scheduling CPU resources for SMP VMs.

Scheduling CPU resources for SMP VMs represents a hard problem, because virtualization changes some of the characteristics of the performance of VMs in ways that the operating system running inside the VM would not expect. There are several proposals for new features that can be added to CPUs or operating systems to help improve the performance of SMP VMs [9,10]. However, there is no existing quantitative analysis and comparison of the techniques used by two of the leading virtualization platforms: Xen and ESX. This thesis provides that analysis and also offers some ideas for additional techniques that can be used to optimize CPU scheduling through dynamic reallocation of virtual CPU resources among different VMs. The thesis is organized as follows: Chapter 2 provides an overview of virtualization technology, and Chapter 3 details the interaction between multiprocessing technology and virtualization. Chapter 4 details the implementation of SMP virtualization in VMware ESX server and Xen and also provides a quantitative analysis of the performance of these two platforms. Chapter 5 discusses some ideas for improving overall system performance through dynamically allocating virtual CPU resources, and Chapter 6 provides a conclusion and discussion of opportunities for future work.

Chapter 2: Virtualization Overview

Virtualization is an idea that has been applied in many different fields of computing to provide improved capabilities in system management and performance. One example of virtualization is virtual memory technology [11], which is present in nearly all modern general-purpose computers. Using this technology, an operating system presents a linear memory address space to applications, and maps virtual memory addresses that the applications use to physical memory addresses that the system uses. This simplifies the task of application developers by allowing them to write programs that use a linear address space without requiring knowledge of the system's physical memory configuration. Virtual memory also allows for paging, which increases the effective amount of main memory available in a computer system. Memory pages that are used infrequently can be copied to disk when not in use, and can be retrieved from disk when needed. Translation of virtual addresses to physical addresses does introduce some overhead for memory accesses. However, the effect of this overhead is minimized by caching the translated addresses in the translation lookaside buffer. Since the benefits of virtual memory far outweigh the costs of implementation, virtual memory is used in nearly all modern general-purpose computers.

Virtualization also plays an important role in the field of computer networking [12]. One example is virtual local area networks (VLANs), which are common in switched Ethernet networks. Ethernet was originally designed as a broadcast network in which communication between any two points on a network segment was broadcast to all systems connected to the network. As the capabilities of integrated circuits improved, switched Ethernet networking was developed. Switched Ethernet supported improved network performance, allowing systems connected to the network to communicate directly through the switch without sending broadcast messages. However, the design of the Ethernet protocol still required broadcasting some messages to all systems connected to the network. VLANs allow large switched Ethernet networks to be segmented into smaller virtual networks. Broadcast traffic is limited to the VLAN on which it occurs rather than being sent to all systems on the network. Although some overhead is also required to support VLANs, the benefits provided far outweigh the costs of implementation, and VLAN technology is used extensively in large Ethernet networks.

The basic idea of virtualization is to hide system implementation details behind a simpler abstract interface. This interface is often the same as the original physical system interface. Virtualization allows new capabilities to be added to a system without changing the interface that the users of the system expect to see. In the case of virtual memory, programs are designed to access memory locations by presenting a physical address to the CPU. Virtual memory allows this design to continue to work unchanged, but behind the scenes the virtual memory addresses are translated to physical addresses. VLANs allow the basic Ethernet addressing design to remain unchanged while providing new capabilities for network segmentation that improve performance and management.

2.1 System Virtualization

System virtualization is simply the concept of abstracting the entire functionality of a computer system as a virtual machine (VM). System virtualization was popular in mainframe computer systems during the 1970s, but the rise of the personal computer in the 1980s seemed to eliminate the primary reason for using virtualization [2]. The availability of inexpensive microprocessor-based computers allowed individual users to have their own machines at a reasonable cost, rather than using a VM on a shared mainframe computer.

As microprocessor capabilities improved, and multiprocessor shared memory computer systems entered the marketplace, operating system design became more complicated and limited the pace of innovation in shared-memory multiprocessor computer systems. In 1997, researchers at Stanford University reintroduced the idea of virtualization as a solution to the problem of writing operating system software for large multiprocessor computer systems [13]. In 1998, these researchers founded a company called VMware to develop and market virtualization products for computer systems based on the Intel x86 architecture. Other companies were also developing virtualization products at the same time; however, VMware's products had the largest impact on the marketplace.

The emergence of system virtualization for the x86 architecture, and its continued growth in popularity, came about from a combination of factors in computer architecture and system software development. As microprocessors improved in performance, their use expanded beyond single-user desktop and laptop computers. Small- to medium-size servers with two to four processors are used in a variety of business applications such as databases, directory services, web servers, and mail servers. The most common system configuration dedicates an entire server to a single application, leading to average system utilization rates as low as 10%. Multiple applications are not usually combined on the same server, even if the server is underutilized, because of the complexity of managing this configuration. But, with system virtualization, multiple VMs can be consolidated on a single server while still segmenting the configuration of the multiple applications and their associated operating systems. Today, server consolidation is one of the primary reasons for the implementation of system virtualization technology.

System virtualization is becoming a pervasive technology in modern operating systems. The term virtualization has become synonymous with system virtualization, despite the fact that virtualization itself is a generalized concept that can be applied to many fields of computing. For the remainder of this thesis the term *virtualization* will be used to refer to *system virtualization*. The rest of this chapter provides an overview of virtualization technology available for the x86 architecture, with particular focus on Xen and ESX. This background material focuses on the common case of virtualizing a uniprocessor system which has only one VCPU. Readers familiar with the virtualization technology used in Xen and ESX may wish to skip ahead to Chapter 3, which begins discussing SMP virtualization.

2.2 Virtualization Marketplace

Today, VMware has the market-leading virtualization technology for systems based on the Intel x86 architecture [14]. VMware's virtualization products are the most mature and are the most trusted by system administrators using general-purpose computers in corporate data centers. However, virtualization is a relatively new technology in the x86 general-purpose computing marketplace, and the use of virtualization technology is expected to increase rapidly in upcoming years. One way that existing virtualization technologies can be categorized is as workstation virtualization products or server virtualization products.

2.2.1 Workstation Virtualization

Workstation virtualization products are usually run as applications in a host operating system, a technique know as hosted virtualization [15]. The development of a hosted virtualization product is simplified because the host OS provides access to hardware devices instead of requiring the virtualization solution to control the hardware directly. The ability to run a separate OS instance in a VM is useful for developers because it allows for easily testing software in an OS that is separate from the one hosting the development tools. If the application is running in a VM, a bug, which causes the OS to crash, will not crash the computer the developer is working on. Another use for workstation virtualization is to run different operating systems on the same computer. For example, it is possible to have a VM running Windows hosted in a Linux OS, in order to provide compatibility with applications that will run only in Windows. VMware has some of the market-leading products for this segment, including VMware Workstation for Windows and Linux, and VMware Fusion for Mac OS X. VMware also has many competitors in this market segment, including Microsoft Virtual PC, Parallels Workstation, and VirtualBox.

Workstation virtualization is a useful technology for developers, but it does not fundamentally change the traditional role of the operating system software. The host OS executes desktop virtualization products in the same way that it runs other application software. The guest OS executing in a VM also executes just as it would if it were controlling physical hardware. Hosted virtualization products are also used for server virtualization in some cases, but this use is generally limited to research and development and test purposes.

2.2.2 Server Virtualization

Bare-metal virtualization is an alternative design in which the virtualization software has direct control of the system hardware, and multiplexes system resources between virtual machines. This design changes the fundamental role of the operating system, and has the potential to significantly alter the system architecture used in general purpose computers. The virtualization software that directly controls the hardware is called the virtual machine monitor (VMM), or hypervisor, and it acts as an intermediary between the operating systems running in guest VMs and the physical computer hardware where instructions are executed. VMware again is the market leader, and VMware ESX Server software is the most successful commercial virtualization software.

One competitor to VMware ESX server is an open source virtualization product named Xen. There are significant differences in the technology used to implement these two software products, and these are covered in detail later in this chapter. However, it is useful to understand the business position of these two different products as it exists today because business use is often a critical factor in technology development.

VMware is the server virtualization market leader, with the most mature technology and the easiest to use tools. VMware's flagship product is VMware Infrastructure, combining the VirtualCenter management tools with ESX server hypervisor in an integrated package that allows for easy deployment and management of VMs. VMware has established an early lead in the virtualization marketplace, and it is likely that their technology will continue to have a significant influence on the industry even as it faces increased competition.

Xen was developed by researchers at the University of Cambridge [5] and released as open source software licensed under GNU GPL in 2003. The company XenSource was founded by some of the researchers who created Xen in order to further its development and commercialize the product. Xen is closely tied to the Linux OS; in the first versions of Xen, Linux was the only guest OS that would run in a Xen VM, and Linux was also required to run the Xen hypervisor. Two major commercial Linux developers, Novell and Red Hat, have embraced Xen and included it in their commercial Linux distributions. Sun has announced an initiative named xVM that will include the Xen hypervisor with the Solaris OS. Oracle has announced a virtualization product named Oracle VM that is based on the Xen hypervisor. While Xen's market share today is small, it is being actively developed and included in virtualization products from many different software vendors.

The final company to consider in the server virtualization marketplace is Microsoft. Microsoft is the leading operating system developer in the general-purpose computing marketplace, and system virtualization has the potential to alter the role of directly controlling system hardware that operating systems traditionally play [16]. Until recently, Microsoft's virtualization products were all hosted virtualization software, which lack the capabilities of bare-metal virtualization and are not widely adopted for server virtualization. However, in June 2008, Microsoft released Hyper-V as an add-on to the Windows Server 2008 OS [17]. This is their first hypervisor-based virtualization solution, and it is sure to have a significant impact on the virtualization marketplace. Microsoft contributed to the initial development of Xen at their Cambridge research lab. Since then they have collaborated with XenSource, Citrix, and Novell in providing support between Xen and Hyper-V. Many of the design decisions made for Hyper-V are similar to the design of Xen.

In the near term, VMware can be expected to continue to dominate the server virtualization market segment. However, server virtualization is a rapidly growing segment, and virtualization products that use the Xen hypervisor, as well as Microsoft's Hyper-V, can be expected to challenge VMware's position as the market leader.

One of the most important uses of system virtualization is for server consolidation. With server consolidation, multiple VMs are run on a single server using virtualization software, and each VM replaces a physical server. Consolidating multiple VMs on a single server can reduce costs for server hardware, supporting infrastructure such as network connections, and power consumption. While cost reduction from server consolidation is attractive, virtualization also reduces the complexity of system administration tasks. It is also possible to dynamically reallocate system resources based on total utilization, reducing the need to allocate extra resources that normally go unused.

2.3 Virtualization History

System virtualization is a technique in which the interface being virtualized is that of a complete computer system. In computer architecture, the ISA represents the well-defined boundary between the hardware implementation and system software programs. The IBM System/360 was the first computer system to use a well-defined ISA, which allowed programs to run on different implementations of the system [18]. The IBM System/370, successor to the System/360, provided the ability to virtualize the System/360 architecture using the VM/370 operating system [19]. The VM/370 was able to run multiple virtual machines (VMs) simultaneously; each VM executed its own operating system that appeared to directly control the hardware, but control was in fact maintained by the virtual machine monitor (VMM). Figure 2.1 depicts a virtualized system with a single physical machine partitioned into three different VMs, each running its own guest OS and applications. The VMM is a software layer between the VM and system hardware, and all VM access to the hardware is controlled by the VMM.

In 1974, Popek and Goldberg defined formal requirements for a virtualizable system architecture [1]. A VM was defined as an efficient isolated duplicate of the real machine, and the VMM was described as software that maintained control of the system. There were three requirements for the VMM: first, program behavior must be unchanged from execution on a real machine; second, a statistically dominate subset of the processor instructions must execute without VMM intervention; and finally, the VMM must maintain complete control of system resources at all times.

While virtualization was included as a feature for mainframe systems in the early 1970s, it was not taken into consideration in the development of commodity microprocessor based

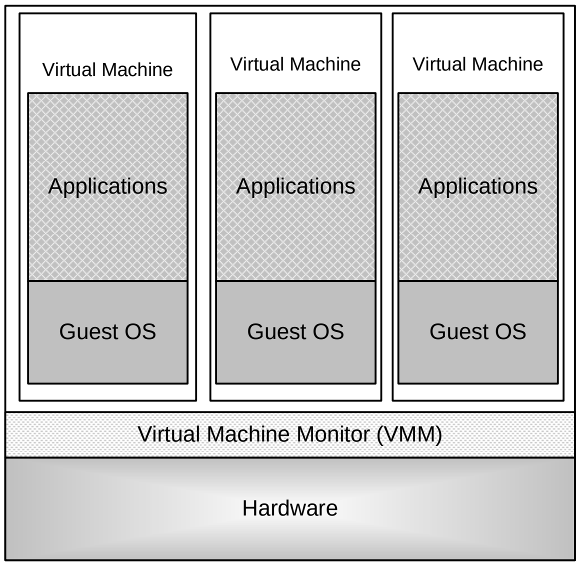

Figure 2.1: System virtualization example

systems such as those based on the Intel IA-32 architecture. Inexpensive microprocessor-based systems seemed to eliminate the need for virtualization because each user received dedicated inexpensive computing resources instead of partitioning expensive mainframe resources among many users. However, as the capabilities of microprocessors increased, there was a corresponding increase in the complexity of operating system and application software running on these systems. By the late 1990s system virtualization was again being considered as a way to efficiently manage the complexity of developing system software for microprocessor-based systems [13].

2.4 Virtualization Requirements

In order for a system to be considered virtualizable it must meet the following requirements [1]:

- Efficiency: Most instructions should be executed by the hardware directly; the VMM should only have to intervene for instructions that affect the global system state.

- Resource control: The VMM maintains control of all system resources at all times. It is impossible for a VM to access any resource without approval of the VMM, and the VMM can revoke resources from a VM at any time.
- Equivalence: Program execution on a VM is indistinguishable from execution on physical hardware, except that timing and resource availability may be changed.

In their 1974 paper, Popek and Goldberg presented a proof of the following theorem:

Theorem 2.1. *For any conventional third generation computer, a virtual machine monitor may be constructed if the set of sensitive instructions for that computer is a subset of the set of privileged instructions.*

A third generation computer has the following features: user and supervisor modes of operations, relocatable memory accesses, ability for non-privileged programs to call privileged system routines, and asynchronous I/O interrupts [3]. Sensitive instructions are those which can modify or reveal some aspect of global system state information, while privileged instructions can only be executed in supervisor mode. To understand why privileged instructions should be sensitive, consider a CPU with an instruction called `SysMem` that reports how much RAM the system has installed. In a virtualized system the total amount of system RAM is partitioned among the VMs running on a system. If the `SysMem` instruction is privileged, when it executes in the VM running in user mode it will cause an exception which the VMM will catch. The VMM can then determine how much memory is actually allocated to the VM and report that value instead of the total system memory. However, if the `SysMem` instruction were not privileged, and the VM could execute it directly, the instruction would report the total amount of system memory rather than the amount the VM was configured for.

The most popular commodity microprocessor-based systems are those that use the x86 architecture; however, the x86 architecture was not specifically designed to support virtualization. An analysis of the x86 instruction set architecture in 2000 revealed that it contained 17 sensitive but unprivileged instructions [3]. As a result, the classical virtualization trap-and-emulate technique, where all instructions are executed directly by a VM and sensitive instructions are trapped and emulated by the VMM, will not work. This is because a VM can execute sensitive instructions that do not cause an exception that the VMM can trap. Although the classical trap-and-emulate virtualization technique does not work on x86 architectures, the techniques of binary translation with direct execution and paravirtualization have been developed to allow the x86 architecture to support virtualization. As virtualization on x86 architecture has become popular, the ISA has been extended with VT-x extensions which allow it to support classical trap-and-emulate virtualization techniques.

2.5 VMware Virtualization

VMware is currently the market leader in providing virtualization solutions for systems based on the popular x86 architecture. VMware was founded in 1998 by students and faculty at Stanford University, and its first product, VMware Workstation 1.0, was released in 1999. The company grew rapidly, and in 2004 VMware was acquired by EMC for $635 million [20]; however, it remained a separate subsidiary of EMC. The incredible success of VMware is due to its ability to deliver a proven, reliable virtualization solution for the x86 architecture.

VMware offers two types of virtualization products: hosted virtualization products such as VMware Workstation, which run as an application under a host OS; and ESX, which controls system resources directly instead of through a lower-level OS. The advantage of the hosted virtualization solution is that it allows reuse of existing device drivers written for the host OS. However, it suffers a significant performance penalty and is only useful for running workloads which do not require high performance. This section focuses on the techniques applied using ESX Server, which allow the VMM to control the physical hardware directly and provide the best performance for VMs. VMware's virtualization technology is best understood by considering the virtualization of three components of system architecture:

CPU, memory, and I/O.

2.5.1 CPU Virtualization

VMware's virtualization implementation for x86 CPUs makes use of the concept of binary translation. This concept was first developed for use in emulation technology, where a binary executable for one architecture is translated into code that can run on a different architecture [21]. However, emulation introduces a significant performance overhead and is not considered virtualization because it fails to meet the efficiency requirement. VMware's implementation of binary translation takes advantage of the fact that it is performing binary translation on the same ISA, which allows the use of a technique where binary translation is combined with direct execution [22].

In VMware's system, the VMM translates the binary code for the VM that is executing into code that can execute safely, while still allowing the VMM to maintain control of the overall system. The translation process has the following properties:

- Uses binary code as input and produces binary output;
- Translation is dynamic, occurs at runtime, and is interleaved with the process of executing the translated code;
- It is a system level translation, no assumptions are made about the guest code, the code produces the same result, including errors, as it would if executed on the physical hardware;
- Input is the full x86 instruction set, while output is a subset that can be executed safely while still allowing the VMM to maintain control of the system;
- Code is generated adaptively in response to guest behavior to improve efficiency.

The translator works by reading guest memory from locations specified by the guest program counter (PC) and decoding the data in this location into intermediate representation (IR) objects, which represent instructions to be executed by the guest VM. The IR

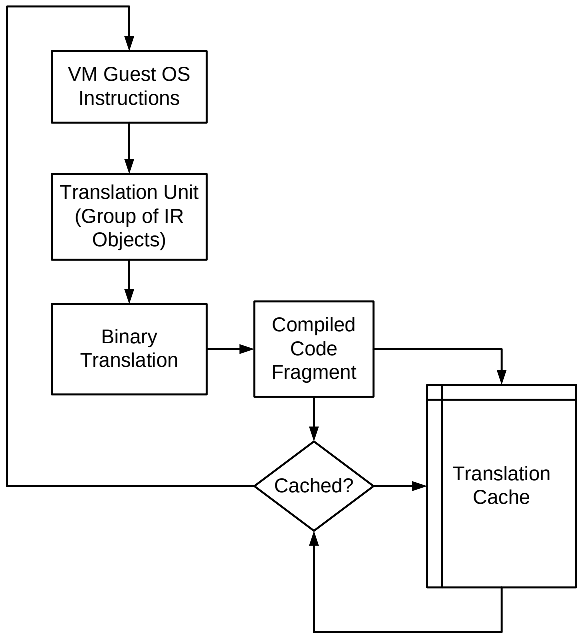

Figure 2.2: Binary translation control flow

objects are grouped into translation units (TU), which are terminated either when they reach an instruction that indicates the end of a basic block (such as a branch instruction), or when they reach a predefined limit of 12 instructions, which is set for performance reasons. The code in a TU is then translated into a compiled code fragment (CCF) that is safe for execution on the physical machine. Figure 2.2 shows the control flow of the binary translation process: the CCFs are cached in a translation cache, and the majority of the time the VM is able to execute cached CCFs. Most of the translated instructions in the CCF will be identical to those in the TU; for example, the addition of values in general-purpose registers `ADD EAX, EBX` does not involve any privileged state and can be translated identically from the TU into the CCF. However, a control instruction such as, `jge LABEL` will need to be translated into two instructions, `jge TRANSLATED_LABEL` followed by `jmp TRANSLATED_FALL_THROUGH`.

The efficiency of the binary translation process is improved through the application of several techniques. First, the CCFs are stored in a translation cache (TC), and as the code executes its working set they will be captured in the TC, at which point the program will execute without calling the translation routines. The second optimization occurs when the guest OS in the VM is executing in user-mode. It is usually safe for the VMM to execute this code directly without performing translation. This is because code executed in user-mode will generate exceptions correctly when performing privileged operations. Another optimization is to modify the translated version of code that causes extensive traps in order to reduce the number of traps that occur. This is done by identifying the code that causes traps and translating this code to link directly to the portion of the VMM that emulates the effect of the trapping instruction.

2.5.2 Memory Virtualization

The guest OS running in a VM believes that it has complete control of the memory assigned to it, starting at address zero and ranging to the maximum value for the memory assigned to the VM. However, the VMM controls allocation of the system's physical memory at the page level and maintains a data structure, called a shadow page table, to map between the physical page numbers and the page numbers presented to the VM [23].

When the guest OS updates its page table to map to a location in the guest's view of memory, the VMM also updates the shadow page table to map the page to a physical location in the machine. When the guest OS accesses a memory location, the physical address used is calculated from the value stored in the shadow page table. Figure 2.3 shows how the guest page table and shadow page table are used to allow the VMM to control access to physical memory. The guest OS has a page table that it uses to map virtual memory addresses to physical memory locations in the VM. While the guest OS uses its page table to generate physical addresses, these addresses correspond to memory locations within the VM, not actual machine addresses. The VMM maintains a shadow page table for each VM and the values in the shadow page table are used to map the VM memory pages to the memory locations in the machine. When a guest in a VM accesses a memory location, the

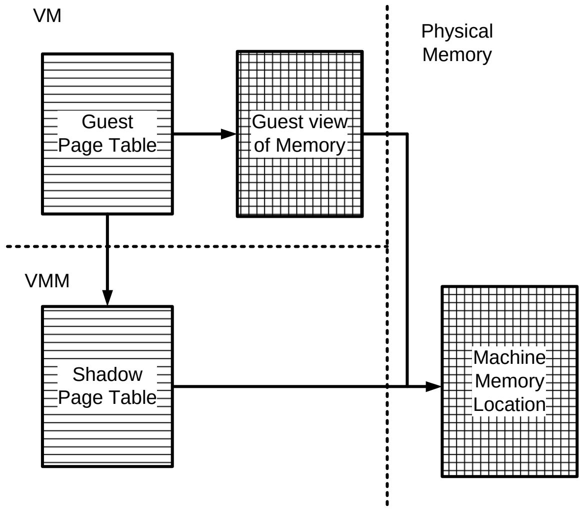

Figure 2.3: Shadow page table operation

VMM uses the shadow page table to redirect this access to the machine memory location where the VMM has mapped the memory.

These techniques provide a relatively straightforward means for the VMM to allocate physical memory to the VMM. However, they do not allow for effective overcommitment of system memory through standard virtual memory techniques. In order to support memory overcommitment, VMware developed the techniques of memory ballooning and transparent page sharing.

Standard virtual memory techniques allow physical memory to be overcommitted by copying pages of memory to disk. While it is possible for the VMM to do this, it lacks the information necessary to intelligently select which page to swap to disk. VMware solves this problem by implementing a balloon driver, which runs in the guest OS and which the VMM can use to request memory from the guest OS running in the VM. The balloon driver requests memory from the guest OS running in a VM and then informs the VMM which memory pages it has received. The VMM is then able to use these memory pages for other purposes, with the knowledge that the VM will not be actively using them.

Multi-user operating systems also have techniques which allow users to share references to read-only memory used for program execution. For example an operating system may load a single copy of a text editor application that is referenced by multiple users simultaneously, reducing the total system memory consumption. In a virtualized environment not only are multiple copies of the same application loaded, but there can also be multiple copies of the same OS. However, the guest OS running in a VM does not have the global system knowledge which would allow it to share memory, and an unmodified guest OS will not load data in a way that is suitable for the VMM to load shared copies of read-only files. VMware solves this challenge by using a technique called content-based page sharing to share identical pages [23]. This hashes the contents of a memory page and compares the hashed values to identify possibly identical pages. Identical pages are marked as copy on write, and can be shared until the system writes to a page.

2.5.3 I/O Virtualization

Efficient virtualization of I/O for the x86 architecture is one of the more complicated tasks because of the wide variety of I/O devices available for the x86 architecture. The approach used by VMware's hosted products, such as VMware workstation, is to convert I/O requests from the guest OS, running on a VM, into system calls from the VMM to the host OS. This virtualization solution provides support for all I/O devices supported on the host OS; however, it introduces a significant performance penalty because each I/O operation requires multiple context switches: first from the VM to the VMM, then from the VMM to the host OS, then from the OS back to the VMM, and finally, from the VMM to the VM [15]. This is acceptable for low-performance devices such as keyboard input, but not for high performance devices such as network interfaces. VMware is able to deliver higher performance with the ESX Server product by having the VMM control the I/O hardware directly, eliminating the overhead introduced by the host OS. This requires device drivers to be written specifically for ESX Server; however, VMware only provides support for a limited number of devices that are provided by major server vendors [2]. VMware also uses a technique where the interface of the driver used by the guest OS is modified to increase its performance. For example, the high-performance `vmxnet` network adapter device driver eliminates many of the I/O instructions that a real network driver would have to perform. Instead, it places the data in a shared memory region and uses a single out instruction to tell the VMM that the data is ready for transmission.

2.6 Xen

While VMware is the market leader in commercial virtualization products, its open-source alternative, Xen enjoys some commercial success and is widely used in academic circles. Xen was developed by researchers at the University of Cambridge and uses a technique called paravirtualization to virtualize the x86 architecture [5]. The Xen source code was made available under the open-source GPL license, and the company XenSource was founded to continue development and market virtualization products based on Xen. As an open source product, Xen has been incorporated into a variety of Linux distributions including commercial products offered by Red Hat and Novell. In August 2007, XenSource was purchased by Citrix for $500 million [24]. XenSource also has a close relationship with Microsoft and is working to optimize performance and compatibility between Xen and future Microsoft operating systems and virtualization solutions. While Xen currently has a small market share, it is a viable alternative to VMware as a high performance virtualization solution.

Xen's paravirtualization is different from full virtualization because the virtual environment is not an exact replica of the physical machine. Instead, the virtual machine presents an idealized interface that the guest OS interfaces with, while allowing the VMM to maintain control of the complete system. The advantage is that it offers the potential for reducing the virtualization overhead, allowing for higher performance systems. The disadvantage is that the guest OS must be modified to support virtualization; however, a relatively small percentage of the OS kernel source code requires modification, and the applications running on the OS do not require any changes. The implementation of the Xen virtualization solution is described by considering CPU, memory, and I/O virtualization.

2.6.1 CPU Virtualization

Virtualization of the CPU requires that the VMM run in a more privileged state than the guest OS. In architectures with only two privilege levels, the guest OS must share the lower privilege level with the applications running on its VM. However, the x86 architecture has four privilege levels, called rings, and ring 0 (most privileged) and ring 3 (least privileged) are the only rings used in contemporary operating systems. Figure 2.4 depicts the overlap of privilege rings: ring 0 is the only ring that can execute privileged instructions. If a privileged instruction is executed in a different ring, the CPU raises an exception and control is transfered to the appropriate exception handler routine. Xen takes advantage of this by modifying the guest OS to run in ring 1, which allows the OS to be isolated from its applications while still allowing the Xen VMM to maintain control of the CPU by running in ring 0. Exceptions are virtualized by maintaining a table with exception handlers that are registered with Xen. These exception handlers are typically identical to the ones used with physical hardware. The exceptions which occur most frequently are system calls and page faults. System calls are optimized by allowing the guest OS to register a fast exception handler that does not require switching to the Xen VMM executing in ring 0. However, this is not possible for page faults because only code executing in ring 0 can read the register that contains the faulting address.

2.6.2 Memory Virtualization

Virtualization of the CPU is the most difficult aspect of the binary translation approach to virtualization used by VMware. For the paravirtualization approach used by Xen, the virtualization of memory is the most difficult implementation aspect, both for the VMM and for the guest operating systems. The x86 architecture has a hardware managed translation lookaside buffer (TLB), as a result, only valid page translations should be in the TLB. Therefore an address space switch usually requires flushing the TLB. Xen deals with this challenge through the use of two techniques: first, guest operating systems manage their own hardware page tables with minimal involvement of the VMM, and second, a portion

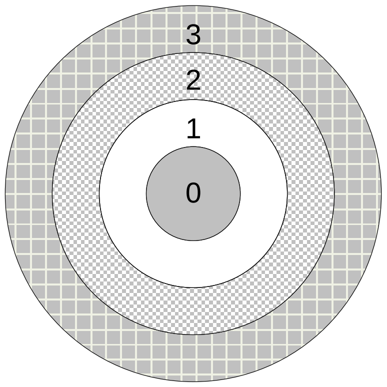

Figure 2.4: IA-32 privilege rings

of the guest OS address space is reserved for Xen. By reserving a portion of the guest OS address space for the VMM, it is possible to switch between the VMM and the guest OS without performing a TLB flush.

The guest OS is able to read from the hardware page tables directly, but modifications must be validated by the VMM before they are allowed to complete. In order to improve the efficiency of page table writes, the guest OS can batch several writes together and allow the VMM to process them at the same time. This reduces the number of context switches required between the VM and the VMM when performing operations with a large number of page table updates.

2.6.3 I/O Virtualization

The paravirtualization technique used by Xen allows it to implement a very elegant solution for device I/O. Rather than have the guest operating systems execute normal hardware device driver code, Xen implements a number of simple device abstractions. Data is then transferred between the guest OS and the VMM using shared-memory, asynchronous buffer rings.

The device drivers that control the hardware are implemented in the VMM. Xen is typically able to reuse the existing device drivers of the host operating system that run in the VMM. This limits the complexity involved in developing Xen, and allows it to support a wide range of hardware devices with high performance.

2.7 Hardware Virtualization Extensions

The popularity of virtualization has led the companies that develop processors for the x86 architecture, Intel and AMD, to introduce processor extensions to help support virtualization. Although this section focuses on the efforts from Intel, the implementations from both Intel and AMD are very similar. The processor and architecture extensions proposed and implemented by Intel are collectively named Intel Virtualization Technology. These technologies range from extensions to the ISA, to new microarchitecture techniques, to proposals to modify I/O and memory communication techniques. Many of these technologies are in the proposal stage and have not yet been fully implemented. However, the ISA extensions for the x86 instruction set have been implemented using technology named VT-x. This section covers the implementation details of VT-x and explains how these extensions are used to improve the capabilities of virtualization implementations.

2.7.1 VT-x overview

The x86 architecture fails to fulfill the requirements of theorem 2.1 because it executes sensitive instructions that are not privileged. Intel's VT-x extensions solve this problem by defining two new modes of operations, VMX-root operation and VMX non-root operation [25]. It also defines transitions between the two modes: a transition between VMX root operation and VMX non-root operation is called a VM entry, and a transition between VMX non-root operation and VMX root operation is called a VM exit. When the processor is executing in VMX-root operation it behaves in a manner similar to a processor without the VT-x extensions. But when the processor is executing in VMX non-root operation, sensitive instructions cause a VM exit to occur, and the processor switches to VMX-root operations. When a VM exit occurs, the state of the processor is saved to a new structure called the virtual-machine control structure (VMCS). These operations support traditional

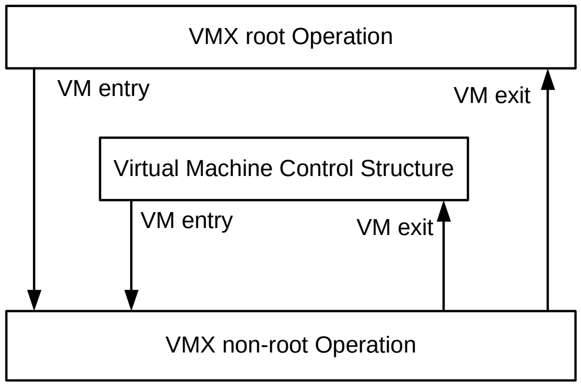

Figure 2.5: Intel VT-x operations

trap and emulate virtualization techniques by having the VMM execute in VMX root operation and the VMs execute in VMX non-root operation. When a VM executes a sensitive instruction, the processor operation is interrupted by the VM exit operation, which allows the VMM to perform the sensitive operation, restore the processor state, and return to the VM using a VM entry operation. Figure 2.5 depicts the flow of control for VM entry and VM exit operations.

While the VT-x extensions allowed the application of classical virtualization techniques on the x86 architecture, they did not immediately provide a performance increase over existing techniques. VMware analyzed the initial implementations of VT-x and found that their binary translation techniques provided as good or better performance for most workloads [22].

2.7.2 VT-x Implementation in Xen

For Xen, the VT-x extensions provided the opportunity to extend virtualization support to guest operating systems that were not modified to support the Xen paravirtualization technique. Xen named this Hardware Virtual Machines (HVM) because they made use of hardware extensions to the x86 architecture instead of modifications to the guest OS to support virtualization. The guest OS executes in VMX non-root operation mode, and privileged instructions result in a VM exit. When a VM exit occurs, Xen performs the operations necessary to simulate the privileged operation, loads the result in the guest portion of the VMCS, and then executes a VM entry [26]. For memory operations, Xen uses a virtualized memory management unit and shadow page table structures, similar to the technique used by VMware. Device virtualization is provided using the open source QEMU emulator. A generic PC platform is presented to the HVM, and when the HVM executes an operation that uses the PC devices, the operation is emulated using the QEMU emulator. However, performance-critical operations were optimized by moving some responses into the VMM as well as modifying the behavior of the emulation model. For example, the graphics emulation model is modified to use a shared memory region instead of causing an interrupt when redrawing the screen.

Chapter 3: Symmetric Multiprocessor Virtualization

3.1 Computer Architecture History and Trends

Modern computers are based on many of the same principles pioneered during the development of the first computers in the 1940s. The name "von Neumann architecture" arose from the ideas that John von Neumann published in his "First Draft of a Report on the EDVAC" [27]. This report described many of the ideas for a stored-program architecture that are in use today. The stored-program architecture reads instructions from memory, accesses data from memory based on information in the instructions, and then accesses the next instruction from memory. This simple design is the basis for most general purpose programming models used in modern computers.

One way to improve performance of a system with a von Neumann architecture is to perform the single operations more quickly — largely equivalent to increasing the frequency at which a processor operates. Another technique is to perform multiple operations simultaneously. Michael J. Flynn developed a taxonomy for categorizing different types of parallel architectures [28], including:

- Single Instruction Single Data (SISD): System executes a single instruction at a time that operates on a single set of data.
- Multiple Instructions Single Data (MISD): System executes multiple instructions simultaneously that operate on the same data.
- Single Instruction Multiple Data (SIMD): System executes a single instruction that performs the same operation on multiple independent operands.
- Multiple Instructions Multiple Data (MIMD): System executes multiple instructions simultaneously that operate on multiple independent data operands.

Of these different classifications, the MIMD categorization is the basis of modern multiprocessor system design.

Another consideration for multiprocessor systems is the interaction between instruction level parallelism and thread level parallelism. Instruction level parallelism takes a single program and finds multiple instructions in that program able to execute simultaneously. Many important innovations in computer architecture are based on maximizing instruction level parallelism, including pipelining, branch prediction, and dynamic out-of-order execution [29]. Thread-level parallelism divides instructions in a program into multiple groups — called threads — and then executes these threads simultaneously.

While exploiting instruction level parallelism has provided an effective way to increase processor performance, it has reached the point of diminishing returns. Today, the microprocessor industry is focused on finding new ways to improve system performance, and one of the most promising techniques is using multiprocessor systems to exploit thread-level parallelism. Performance improvements related to instruction level parallelism mostly have been handled automatically by the internal working of microprocessors, or by built-in compiler optimizations. Programmers for general purpose computers have been able to program in a sequential style suited for the 1940s-era von Neumann architecture. To exploit thread-level parallelism, however, requires a change in the programmer's coding style: instead of allowing the system to automatically parallelize instruction execution, a programmer must explicitly determine which sections of a program can execute in parallel, and include synchronization instructions to control execution of instructions that must occur in a sequential order. Finding ways to make it easier to develop multithreaded parallel applications is a subject of extensive research [30].

Microprocessor developers have firmly embraced the path of supporting increased levels of thread-level parallelism in modern computer architectures, even as software developers have yet to fully embrace the changes required to utilize these resources. On-chip support for thread-level parallelism comes in two primary forms: chip multiprocessors (commonly called multicore processors), and simultaneous multithreading (SMT). Chip multiprocessors take the basic processing element in a CPU and replicate it two or more times on a single socket. Today, four-core chips are common, and that number can be expected to increase rapidly in the future. SMT takes the resources of a single processor core and partitions those resources between multiple threads that can execute at the same time. The advantage of SMT over multicore processors is that only a small portion of the processing core needs duplicated structures, and the overall system performance gain can exceed the cost in transistors and power consumption for the duplicated structures [31]. Intel is planning to introduce microprocessors on the x86 architecture that will combine multicore and SMT technology. Sun already sells the UltraSPARC T2 (code name Niagara 2) that combines multicore technology with SMT to allow for up to 64 threads to execute simultaneously on a single chip [32].

3.2 Multiprocessing

Multiprocessing has only recently begun to dominate the general purpose computing landscape, but it has a long history in the field of supercomputing, also known as high performance computing. In June 2008 the world's top-ranked supercomputer was the IBM "Roadrunner," specially built for the Los Alamos National Laboratory [33]. This system combines 6,562 dual-core AMD Opteron chips and 12,240 IBM Cell processors in a cluster configuration with 122,400 processing cores. This type of system is used primarily for scientific experiments such as molecular modeling and weather simulations. The programs written for these systems are highly parallel and make extensive use of multithreaded techniques.

One of the challenges in developing a multiprocessor computer system is determining how to interconnect the processors and memory. Another challenge is partitioning the computing work between the processing elements. Supercomputers such as the Roadrunner have complex strategies to interconnect the processors and memory and to divide the system workload between processing elements. The Roadrunner is an example of asymmetric multiprocessing, not only because it has two different types of processors that handle different tasks, but also because the IBM Cell processor itself has different types of processing cores on the chip.

3.2.1 Symmetric Multiprocessing

General-purpose multiprocessor computers usually take a simpler multiprocessing approach called symmetric multiprocessing (SMP), where any processing element can execute any tasks in the system. In an SMP system, all processors are identical and no processor has a separate dedicated special purpose as is the case in an asymmetric multiprocessing system.

Until recently, general purpose multiprocessor systems also used a shared bus to interconnect the processors and memory. All intercommunication between processors took place on the shared bus, as well as all communication from the processors to memory. The advantage of the shared bus architecture was its simplicity; the disadvantage was that it could quickly become a system bottleneck as more processors were added. Most general-purpose multiprocessor systems are limited to two or four processors. Individual systems with more processors would usually require techniques associated with HPC to interconnect the processors. However, as the number of cores per processor has increased, a shared bus architecture has begun to limit the performance of even dual socket architectures. AMD has already introduced a direct connect architecture called “hypertransport” [34], and Intel is planning to introduce a similar architecture named “Quickpath.” Symmetric multiprocessing is supported by most general purpose operating systems, including Microsoft Windows, Linux, and Sun Solaris.

3.2.2 SMP Workloads

Chapter 2 discussed the history of virtualization technology as well as the implementation details for some of the popular virtualization systems used with the x86 architecture. The increased interest and popularity of virtualization can be attributed to many factors, but perhaps most important is the fact that virtualization helps improve utilization efficiency for modern multicore processors used in multiprocessor servers. Often, virtualization technology is adopted in response to server consolidation workloads.

While parallel computing has a long history in HPC, only recently has it become popular in general purpose computing. Multiprocessors require the programmer to explicitly synchronize which activities will occur in parallel. But most programs for general purpose computers are still written as a set of sequential actions — targeted for a computer architecture using the same programming paradigm that von Neumann described in the 1940s.

One workaround is server consolidation, which uses virtualization to allow multiple single threaded programs to run safely on a single host system. Although all modern operating systems support multitasking (the ability to execute multiple programs at the same time), usually it is not practical to combine multiple business applications together on the same server. Many applications are closely tied to the operating system, and it is impractical to configure certain dissimilar applications to run in the same operating system instance. Furthermore, many operating systems do not provide any way to limit the resource utilization of a single process. System administrators configuring load-balanced and redundant systems have adopted the practice of installing separate applications on different servers. However, as processors and systems have become more capable, the utilization of individual servers has dropped to the point where average utilization is often well under 10% of system processing resources.

Virtual machines provide a way for system administrators to dedicate specific guaranteed resources to applications, yet combine them on the same server. This is done through configuring a VM with the desired OS and applications and then combining multiple VMs on a single physical host system. Known as server consolidation, this is one of the most important use cases for virtualization.

3.3 SMP Virtual Machine Challenges

Although the preceding section raised concerns about the availability of multithreaded applications in the multicore era, there are many instances of multithreaded applications running in common, general purpose computing environments. VMs are designed to replicate the characteristics of the physical machine that they replace, and just as physical machines can support SMP, so too can a VM support virtualized SMP. Virtualized SMP is simply the assignment of multiple VCPUs to a single VM. The guest OS treats each of the multiple VCPUs in an SMP VM just as it would a physical CPU in a physical system.

As described in Chapter 2, virtualization shares the resources of a single host system with multiple VMs. One of the shared resources is CPU time; however, virtualization introduces some differences in the timing characteristics for scheduling VCPUs compared to the behavior of a physical system. On a physical system, the operating system has direct control of the system hardware and can choose when and where processes are scheduled on CPUs. In a VM, however, the guest OS only controls the scheduling of processes on VCPUs, while the VMM controls the scheduling of VCPUs on physical CPUs.

This can be problematic, because the guest OS running in a VM schedules processes to run on VCPUs with the expectation that the process will execute on a physical CPU at the time the guest schedules it on a VCPU. However, the guest OS only controls scheduling on VCPUs, while the VMM controls scheduling of VCPUs on physical CPUs. An SMP VM may have synchronization routines that it executes, and these synchronization methods may exhibit poor performance if one VCPU is running while another one that it depends on is not running.

3.3.1 Spinlocks

The most obvious example of this problem is a synchronization method known as a spinlock. Spinlocks are commonly used for very fast synchronization between two threads that are known to be executing simultaneously on different processors in a multiprocessor system. In this case it can be more efficient for a thread to execute on one processor continually and check the status of some synchronization primitive while waiting for a thread executing on a different processor to set the synchronization, indicating that the first thread can continue executing. The process "spins" while waiting for a "lock" to be released. In many cases this results in better performance and better CPU utilization than using a more complicated synchronization method that would require a context switch.

However, if assumptions that underlie the implementation of a spinlock are violated, then the spinlock can waste CPU resources, or even cause a crash in the guest OS. For example, suppose one VCPU is scheduled on a physical CPU and is waiting for a spinlock to be released, but the lock that it is waiting on is held by a VCPU not currently scheduled. Instead of the spinlock serving as an effective synchronization method, the VCPU executing the spin loop could spin for its entire time-slice until it was preempted by the VMM. In the best case this would be a very inefficient use of processing resources, and in the worst case it could violate OS timing constraints and cause the OS to crash.

3.3.2 Co-scheduling

One way to avoid this problem is to co-schedule the VCPUs that are part of an SMP VM. With co-scheduling, either all of the VCPUs in a VM are executed simultaneously, or none of them are. This solves the problem with spinlocks and other synchronization methods, but it introduces a new problem for efficiently using CPU resources. First, only some of the VCPUs in an SMP VM might have work to do at any point in time. If all of the VCPUs have to be scheduled simultaneously, including those that are idle, this results in less efficient CPU utilization than if idle VCPUs could simply remain idle. Second, it reduces the flexibility of the VMM for scheduling VCPUs with physical CPUs, because there must be enough idle physical CPUs to simultaneously schedule all the VCPUs in an SMP VM. Consider the situation where a host has four physical CPUs and is executing two virtual machines: one with four VCPUs, and the other with only one VCPU. All four physical CPUs must be idle in order for the 4-VCPU VM to be scheduled, and when the 1-VCPU VM is executing, three of the physical CPUs must be idle even if the VCPUs in the 4-VCPU VM have work to do. If there were no need to synchronize the VCPUs in the 4-VCPU VM, each of them could be scheduled independently, and overall system utilization could be improved.

3.4 Proposed optimizations for SMP VM scheduling

Scheduling of VCPUs in an SMP VM is a challenging problem, and Chapter 4 provides an analysis of approaches taken by two different real virtualization systems, Xen and VMware ESX server. This section presents a survey of two optimizations proposed to address the problem of spinlock synchronization in virtualized systems.

3.4.1 Avoiding Lock-Holder Preemption

The first proposal, by Uhlig et al., is to avoid lock-holder preemption [9, 35]. Two different mechanisms for achieving this are suggested. One is an intrusive approach that requires modifying the guest OS and would be best suited for a paravirtualized system. The other is non-intrusive and could be used in a fully virtualized system.

The first approach is for the guest OS to give the VMM hints about its behavior so that the VMM can avoid preempting a VCPU that is holding a lock. This is done by having the guest indicate that it should not be preempted for the next n microseconds, before it acquires a lock. After the guest OS releases the lock, it will indicate to the VMM that it is again acceptable for the VCPU to be preempted. If the VMM determines that it needs to preempt the VCPU, it will delay doing so during the n microsecond timeframe. If the guest OS has not released the lock by the end of that timeframe, it will be preempted.

The second approach is to divide the execution time of a VM into *safe* and *unsafe* states. The safe state is defined as when the system is executing in user-mode, because all kernel locks will be released before the guest OS transitions from kernel mode to user mode. The safe state also includes when the VCPU is executing the idle loop. The unsafe state is defined as any time the guest OS is executing in kernel mode, because it is possible that kernel-level locks may be held in this state. The VMM is configured to only preempt a VM that is in a safe state. An additional optimization is that a special driver can be added to the guest OS that allows the VMM to force this guest OS into a safe state. This could be necessary if the VMM needed to preempt a VCPU but the guest OS never allowed it to enter a safe state.

The authors note that their proposals are only useful for applications without strong cross processor scheduling requirements. Applications requiring close communication between multiple processors will not necessarily benefit from their approach. The SPEC OMP2001 benchmarks used in the analysis in Chapter 4 are an example of this type of application.

3.4.2 Hardware Support for Spin Management

The second proposal is to add additional hardware support to a microprocessor to allow it to detect when a VCPU is executing a spin-loop [10, 36]. If a VCPU is detected executing a spin-loop, then that VCPU is a candidate for preemption by the VMM. This approach can be considered the opposite of the approach described in the prior section. The previous approach attempted to avoid preempting VCPUs that were holding locks, which should consequently prevent other VCPUs from executing excessively long spin-loops waiting for a lock to be released. This proposal takes the approach that a lock holder could be preempted; however, the VCPUs that execute spin-loops waiting for that lock will be preempted as well. This increases the likelihood that the VCPU holding the lock will be scheduled, and it also reduces the amount of wasted work in the system.

This technique requires additional hardware which is not present in any current processor. The authors note that when a process is executing without making forward progress, the system state will be largely unchanged. They propose adding a CPU counter to check for k store instructions for every N committed instructions. With fewer than k stored instructions, the system decides the process is not making forward progress and identifies it as executing a spinloop and, thus, a candidate for preemption.

The proposed system was implemented using a full system simulator, and in some benchmarks the proposed hardware support resulted in a 10-25% speedup over gang scheduling. However, there have not been any announced plans to add support for spin detection to commercial microprocessors, so for now this proposal remains limited to providing theoretical results.

Chapter 4: SMP Virtualization in ESX and Xen

The previous chapter provided background on the increasing importance of SMP in general purpose computing and discussed some of the difficulties involved with implementing virtualized SMP. Co-scheduling VCPUs is the most straightforward way to implement SMP virtualization, and the proposals surveyed in Chapter 3 were compared with co-scheduling. However, neither ESX nor Xen uses the simple co-scheduling implementation as described in Chapter 3. This chapter provides a quantitative analysis of the implementation of SMP virtualization that is used in real systems, and draws conclusions about the impact of these design decisions.

4.1 CPU scheduling background

Conceptually, the purpose of CPU scheduling by the VMM is similar to CPU scheduling that is done by an OS running multiple processes [37]. The purpose of a scheduling algorithm is to increase the efficiency of system resource utilization by allowing multiple processes to appear to execute simultaneously on the same CPU. One important consideration for CPU scheduling is the need to balance the performance of CPU bound and I/O bound processes. In a mulitprocessor, system another consideration is load balancing processes between processors. For scheduling VMs, the VMM divides CPU resources into time slices and allocates CPU time to specific VCPUs associated with VMs.

4.1.1 ESX scheduling

The scheduling algorithm used in ESX is designed to minimize virtualization overhead and provide high performance for VMs. It also seeks to provide a good balance between the performance of VMs that are executing I/O intensive workloads and those that have CPU intensive workloads. In ESX there are three parameters for configuring CPU priority for a VM:

- Shares: Controls the relative allocation of CPU resources between VM. For example, if one VM has a share value of 1000 and another has a value of 2000, then the one with 2000 can receive twice as much CPU time.
- Reservations: A minimum fraction of host's total CPU time that is guaranteed to a specific VM.
- Limit: The maximum fraction of host's total time that a VM can receive.

The scheduling algorithm used in ESX also has optimization techniques for NUMA (non uniform memory access) systems and for systems that use hyper-threading. NUMA systems divide the memory into nodes, and while all CPUs can access all memory, the latency varies depending on which node is accessed. Hyper-threading is a specific implementation of simultaneous multithreading, allowing a single processor to appear as two logical processors. The details regarding how these technologies impact CPU scheduling are beyond the scope of this thesis, and neither of these technologies was used in the experiments performed for this thesis. For more details see [38].

In ESX, the implementation of SMP virtualization is called virtual SMP. It allows a VM to be configured with either 2 or 4 VCPUs, in addition to the standard configuration of 1 VCPU. VMware introduced virtual SMP in ESX 2.0; it was limited to 2 VCPUs and used strict co-scheduling to ensure predictable performance for VMs using virtual SMP. With strict co-scheduling, all of the running VCPUs in a VM are required to execute simultaneously.

The ESX scheduler implements co-scheduling by keeping track of a skew value between VCPUs [39]. A VCPU accumulates *skew* if it is not running and another VCPU in the same SMP VM is running. If the skew value exceeds a certain threshold, then both VCPUs are descheduled and can only be restarted when they will be able to run simultaneously. As an optimization, ESX has a mechanism to detect VCPUs that are executing the idle loop;

these VCPUs do not add to the skew value and do not need to be co-scheduled with other VCPUs in an SMP VM.

In ESX 3.x, virtual SMP was changed to allow for either 2 or 4 VCPUs, and the scheduling algorithm was enhanced to use relaxed co-scheduling. Relaxed co-scheduling still uses the skew value, but it does not require all VCPUs in a VM to be started simultaneously — only those that have exceeded the skew threshold must start simultaneously.

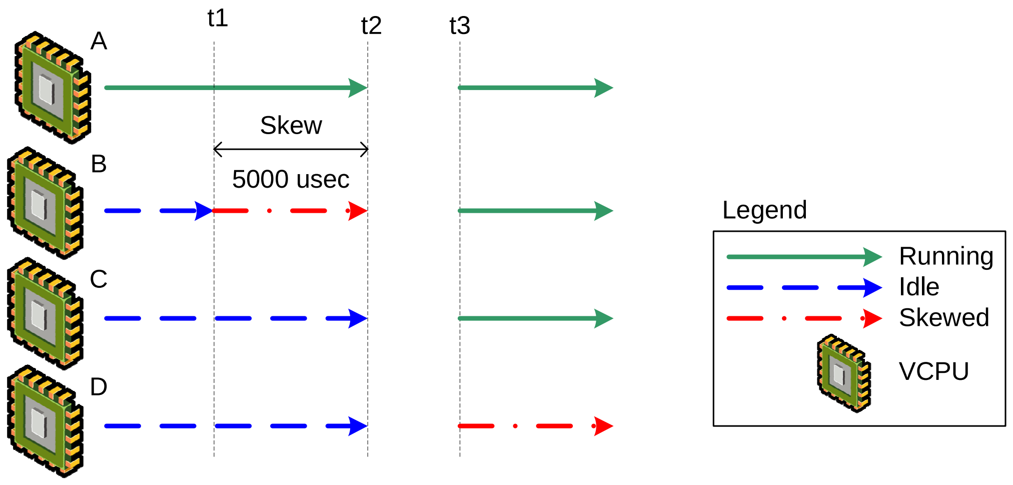

Figure 4.1: ESX scheduling example

Figure 4.1 shows an example of how virtual SMP scheduling works in ESX. There are four VCPUs — labeled A, B, C, D — that are in a 4-VCPU SMP VM. Initially, VCPU A is running while VCPU B, C and D are idle. At time t1, VCPU B becomes runnable, but the VMM is not able to schedule VCPU B to run on a physical CPU. As a result, VCPU B starts to accumulate skew. The skew is the time that VCPU B is not scheduled to run while it is runnable and another VCPU in the VM is running. At time t2, VCPU B has accumulated the maximum allowable skew of 5000 microseconds. As a result, VCPU A is also *forcibly descheduled* by the VMM to prevent VCPU B from accumulating additional skew. When VCPU A is forcibly descheduled, the VMM interrupts its execution to prevent it from continuing to run. After this happens, none of the VCPUs in the VM are running on a physical CPU in the host.

Later, at time *t3*, all 4 VCPUs are runnable, and in this example the VMM is able to schedule 3 of the VCPUs to run simultaneously. Because VCPU B has accumulated the maximum allowed skew, it must run on a physical CPU in order for any other VCPU to be able to run. If strict co-scheduling were used, then all 4 VCPUs would have to wait, and 3 physical CPUs would be idle waiting for a 4th physical VCPU to be available so that all 4 VCPUs could execute simultaneously. However, this example shows relaxed co-scheduling used in ESX 3.x, and VCPUs A, B, and C can start executing while VCPU D starts to accumulate skew. If the VMM is able to schedule VCPU D before it accumulates a maximum amount of skew, then all the VCPUs in the VM will continue to run. Otherwise, all the VCPUs will be descheduled.

Several parameters can be configured to adjust the performance of SMP VM scheduling:

- CPU.SkewPollUsec: interval between coscheduling skew checks. Default value 1000 microseconds
- Cpu.SkewInstanceThreshold: maximum individual skew between VCPUs. Default value disabled
- Cpu.SkewCumulativeThreshold: maximum cumulative skew between VCPUs. Default value 5000 microseconds

In section 4.3 the performance of ESX server is analyzed, and the results show that for CPU intensive workloads the SMP CPU scheduling algorithm performs very well.

4.1.2 Xen CPU Scheduling

As an open source project, Xen has a very different development model from that used by VMware for ESX. One of the differences is that Xen supports multiple different scheduling algorithms and the source code is written so that it is relatively easy to add a new scheduling algorithm to Xen. There have been several scheduling algorithms supported through Xen's history; however, the current default scheduler is the *credit scheduler*. This scheduler was introduced as a new scheduler option with an update to the Xen development version accompanied by a message sent to the xen-devel mailing list in May 2006. Within a month the development version of Xen was configured to use the credit scheduler as the default. For Xen 3.2, the credit scheduler is the default scheduling algorithm for the open source distribution of Xen, and is the scheduling algorithm analyzed in section 4.4.

The credit scheduler is a proportional share scheduler that is work conserving by default. A proportionate share scheduler allocates CPU time in proportion to the number of shares that a VM is assigned. A work-conserving scheduler is one that will only have idle CPU resources if there are no VCPUs waiting to be scheduled. In contrast, a non work-conserving scheduler can have idle CPU resources if the maximum CPU utilization for specific VMs is limited. By default the credit scheduler does not limit the CPU use of a VM; however, a cap value can be configured to limit CPU utilization.

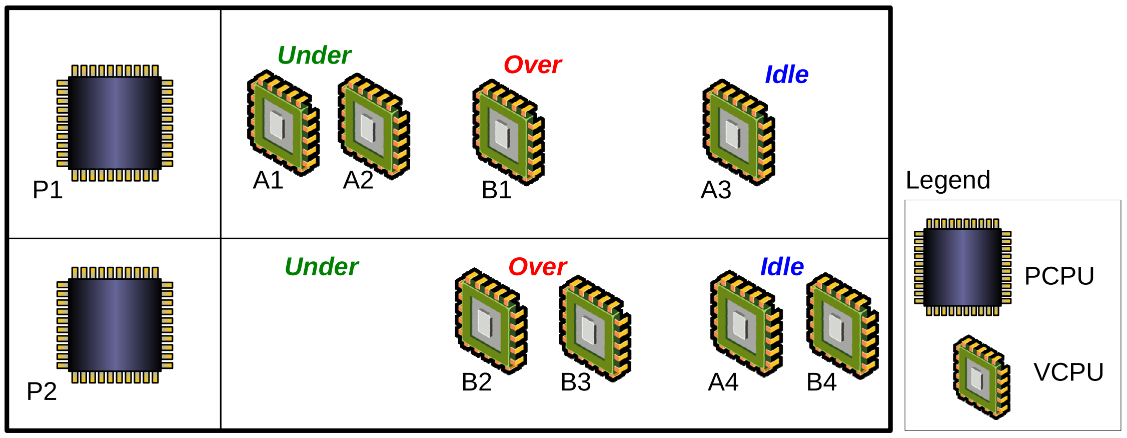

Figure 4.2: Example of CPU scheduling algorithm used in Xen with 2 PCPUs (P1 and P2) and 2 VMs (A and B), each with 4 VCPUs. The runqueues extend horizontally to right of the PCPUs. VM A has remaining credits and its VCPUs have priority *under*, while VM B has used all its credits and its VCPUs have priority *over*. Shows initial state with two PCPUs waiting to make scheduling decisions.

Figure 4.2 shows an example of how the credit scheduler algorithm works in Xen. In contrast to ESX, the Xen credit scheduler does not attempt to co-schedule VCPUs; however, it does do load balancing between physical CPUs in an SMP system. The example in figure 4.2 shows 2 physical CPUs — labeled P1 and P2 — and 2 VMs, each with 4 VCPUs. The two VMs are A and B, and their VCPUs are A1, A2, A3, and A4 for VM A, and B1, B2, B3, and B4 for VM B. Each PCPU has an associated runqueue which holds the VCPUs assigned to that PCPU, and is represented by the VCPUs shown to the right of the PCPU. The VCPUs are shown in their order in the runqueue, with the leftmost VCPU at the head of the runqueue.

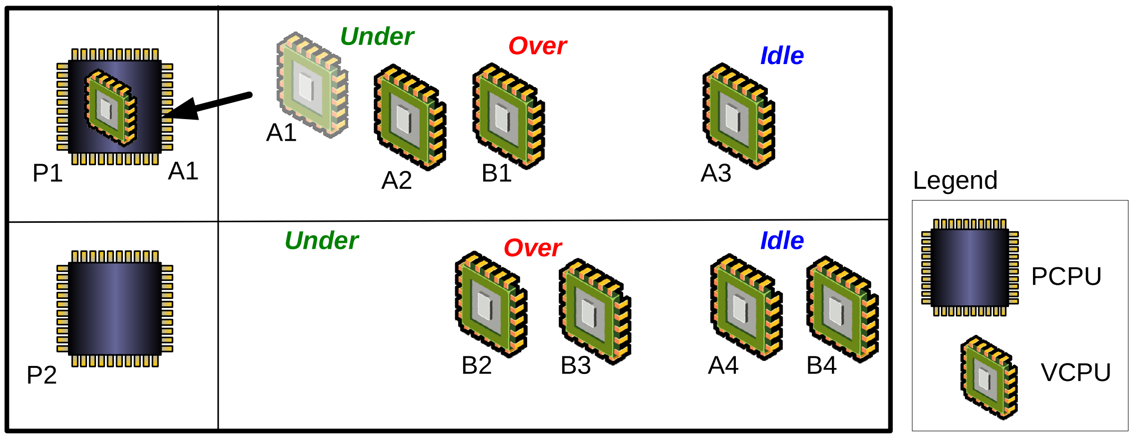

Figure 4.3: Scheduling decision made by PCPU P1. P1 selects the first VCPU in its runqueue, A1, and starts running it immediately because it has priority *under*.

In this example, VM A has credits remaining, causing its running VCPUs to have a priority of *under*. VM B has used all its credits, so its VCPUs have a priority of *over*. The VCPUs that are not runnable have a priority of *idle*. In this example, both P1 and P2 are making a scheduling decision to determine which VCPU to run, and P1 will make its decision first.

As shown in figure 4.3, P1 will select the first VCPU in its runqueue, which is A1. Since A1 has a priority of *under*, P1 will start running it immediately. Figure 4.4 shows how P2 will select the first VCPU in its runqueue, which is B2. Since B2 has a priority of *over*, P2 will check P1's runqueue for any higher priority work to be done on the host. P2 will select A2, and since A2's priority is *under*, P2 will remove A2 from P1's runqueue, and P2 will begin executing A2.

This is a fast and efficient algorithm that dynamically balances VCPUs between PCPUs in an SMP system, and is able to support scheduling for a large number of VCPUs. However,

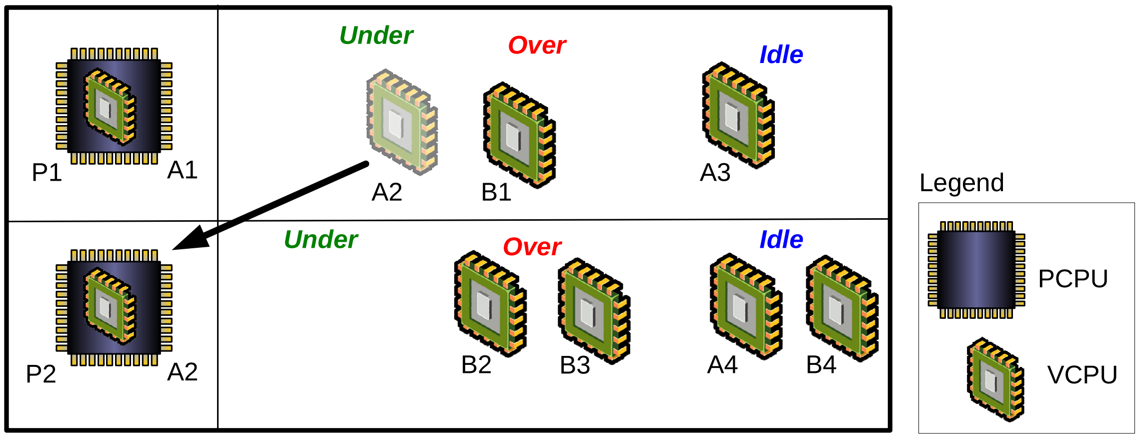

Figure 4.4: Scheduling decision made by PCPU P2. P2 selects the first VCPU in its runqueue, B2, which has a priority of *over*. P2 then checks P1's runqueue for any higher priority work; in this case there is, so P2 removed A2 from the P1 runqueue and starts running it.

it does not have any logic for co-scheduling the VCPUs in a VM — all VCPU scheduling decisions are made independently.

4.2 Experimental Setup

4.2.1 System Configuration

ESX and Xen were analyzed using benchmarks running on a common hardware platform configured in a dual-boot configuration. The software configuration for the guest VMs was also kept as similar as possible in order to limit differences to the virtualization system. The specifications for the system hardware that was used are shown in table 4.1. The 2 quad-core CPUs provided a total of 8 logical processors.

The benchmarks performed were selected to stress the CPU as much as possible in order to evaluate the effectiveness of the CPU scheduling algorithms. The performance of the hard drive and network adapter had no impact on the system performance evaluation. However, the memory performance is closely tied to the performance of CPU benchmarks, and could be expected to influence the results of system benchmarks in some cases. The configuration for the software used for testing ESX is shown in table 4.2, and the software configuration for testing Xen is shown in table 4.3.

Table 4.1: Test system hardware configuration

System	PowerEdge 1950 Server
CPU	2 x Quad Core Intel Xeon X5355 2.66 GHz
Memory	16 GB 667 MHz RAM
Disk drive	4 x 146 GB 10,000 RPM SAS drive

Table 4.2: VMware ESX software configuration

Virtualization System	VMware ESX Server 3.5.0 (build 64607)
Guest OS	Ubuntu 7.04
Guest kernel version	Linux 2.6.20-15
Compiler	Intel C++ compiler 10.1 20080312

Table 4.3: Xen software configuration

Virtualization System	Xen 3.2.0 (dom0 Linux 2.6.18.8-xen)
Guest OS	Ubuntu 7.10
Guest kernel version	Linux 2.6.18.8
Compiler	Intel C++ compiler 10.1 20080312

4.2.2 Benchmarks

The Standard Performance Evaluation Corporation (SPEC) is a non-profit organization founded in 1988 that develops standardized benchmarks for evaluating performance across a variety of systems [40]. The SPEC methodology combines common application source code and input data in a test suite to objectively measure system performance across multiple platforms.

For this thesis, benchmarks from two SPEC benchmark suites were used to evaluate the performance of SMP virtualization in ESX and Xen. First, the SPEC CPU2006 benchmark suite was used for single threaded integer code. A single benchmark test was used in order to allow for consistency executing this benchmark with multiple virtual CPU configurations. The 445.gobmk benchmark was selected because of its low memory requirements, which made it possible to run multiple iterations of the benchmark simultaneously in multiple virtual machines. The 445.gobmk benchmark is based on a program that plays the game Go; the benchmark uses the program to analyze selected Go positions, and can be considered an artificial intelligence application.

The second benchmark suite used was the SPEC OMP2001, selected in order to measure the performance of multithreaded programs. The OMP2001 benchmark suite was developed to test the performance of systems executing parallel applications on shared memory SMP systems. The OMP2001 benchmark suite consists of selected floating point applications from SPEC CPU2000 with added OpenMP compiler directives to define sections of code that can execute in parallel [41]. OpenMP can be added to an application to define a region of code that can be executed in parallel. This simplifies the task of writing parallel applications because some of the work for creating and managing threads is handled by the compiler.

Multithreaded programs are one of the primary ways that the capabilities of SMP systems can be used effectively. However, synchronization demands of multithreaded programs are one of the most difficult problems to solve using virtualized SMP. The 332.ammp benchmark was selected from the OMP2001 benchmark suite and used to analyze the performance of SMP VMs executing multithreaded code. The 332.ammp benchmark was selected because it demonstrated a good speed-up factor in the baseline configuration on the Dell server used to conduct the tests. An ideal speed-up for a multithreaded application changing between running on a 1 CPU system and an 8 CPU system would be 8. However, few real-world applications will be able to achieve a linear speed-up, due to communication required between threads and the need to execute non-parallel sections of code sequentially. The 332.ammp was able to achieve a 6.5 speedup when going from 1 CPU to 8 CPUs in a single VM executing in Xen, and as a result it seemed a good candidate for more complex tests. The 332.ammp benchmark is based on the ammp floating point benchmark taken from the SPEC CPU2000 benchmark suite. The benchmark was modified with OpenMP directives to indicate regions of code suitable for parallelization. The ammp application is a molecular mechanics, dynamics and modeling program.

The remainder of this chapter consists of graphs demonstrating the results of benchmark execution. Results from the benchmark execution have been correlated and combined in graphs comparing relative execution runtime. Most of the graphs are normalized to the runtime of one of the values in the graph in order to make the relative performance easier to visualize. In these cases, lower values on the graphs indicate better performance because a lower value is equivalent to a shorter runtime.

4.3 ESX Benchmark Results

4.3.1 Single Threaded Benchmarks

VMware ESX uses a very different scheduling algorithm from Xen, and it is also a much more conservative system. The open source version of Xen supports up to 32 VCPUs per VM [42], while ESX supports only 1, 2, or 4 VCPUs per VM. This limited the number of VM configurations available for testing in ESX. The ESX SMP scheduling algorithm also uses co-scheduling, where it executes all VCPUs in a VM at the same time.

The first benchmark executed in ESX was a single instance of the single threaded 445.gobmk benchmark executing in a single VM with different numbers of VCPUs. This benchmark was included to provide a performance baseline for an underutilized host system. The results are normalized to the 1 VCPU configuration, and the results in figure 4.5 show that for a host with excess CPU resources, adding extra VCPUs to a VM does not introduce any noticeable performance overhead.

For the second ESX benchmark, the system was tested with 8 VMs running simultaneously, each executing a single instance of the 445.gobmk benchmark. A similar experiment was executed in Xen, the primary difference being that ESX supports a maximum of 4

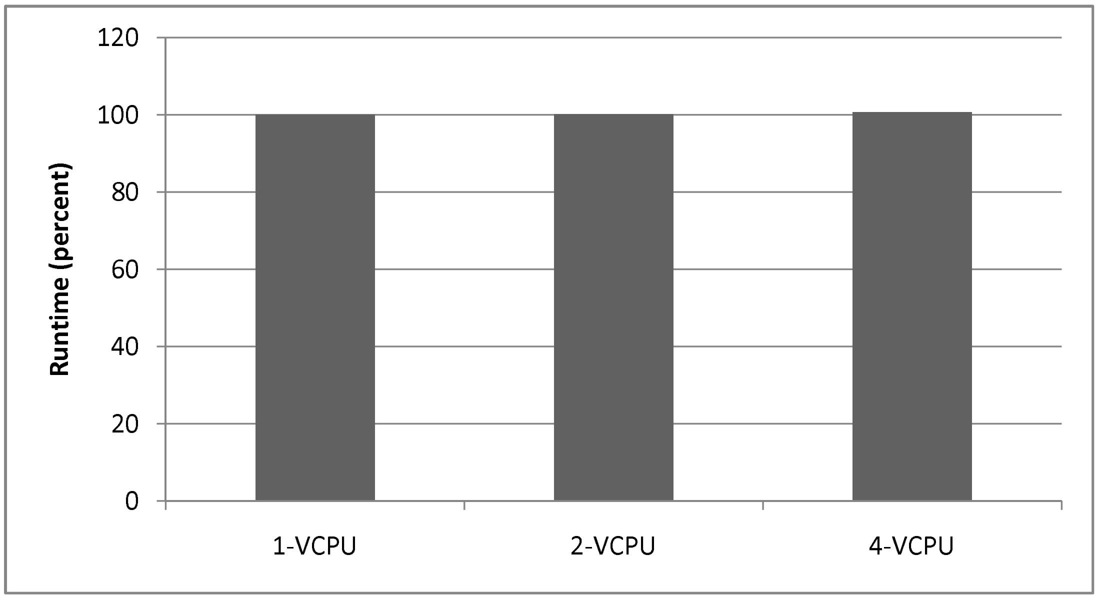

Figure 4.5: Results for 1 VM executing 1 instance of 445.gobmk.

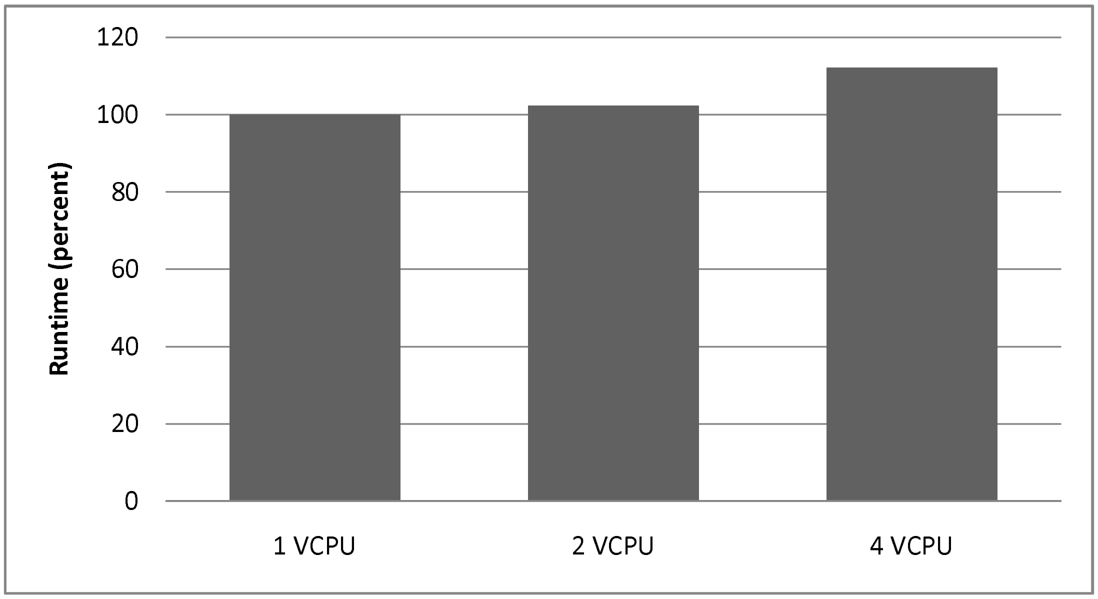

Figure 4.6: Results for 8 VMs simultaneously executing 1 instance of 445.gobmk.

VCPUs per VM. The results for this test, shown in figure 4.6, display a measurable performance overhead as the number of VCPUs per VM increases. The results are normalized to the 1 VCPU configuration, and the 2 VCPU configuration takes 2.4% longer to execute the benchmark, while the 4 VCPU configuration takes 12.2% longer to execute the benchmark. This is a managable level of overhead; however, Xen has a negligible performance overhead of less than 0.2% for a 4 VCPU VM in the same test. ESX's use of co-scheduling does not provide the nearly no overhead type of performance that the Xen credit scheduler provides. Futhermore, ESX is limited to a maximum of 4 VCPUs per VM, whereas Xen supports up to 32 VCPUs per VM. If the performance overhead of the ESX co-scheduling algorithm scaled linearly, it would have over 30% overhead for an 8 VCPU system.

4.3.2 Multithreaded Benchmarks

The next set of ESX benchmarks was designed to test the performance of the multithreaded applications. A similar experiment was also performed with Xen; the difference is again that ESX is limited to 1, 2, and 4 VCPUs per VM. The first experiment had a single VM executing the 332.ammp_m benchmark with varying numbers of VCPUs. The results for this benchmark are shown in figure 4.7 and are normalized to the 1 VCPU configuration. Adding additional VCPUs results in a nearly linear performance improvement.

In the second experiment, 8 VMs with varying numbers of VCPUs simultaneously executed the 332.ammp_m benchmark. The results are shown in figure 4.8, again normalized to the 1 VCPU configuration. These results are quite good because ESX experiences only a moderate slowdown of 2.9% for the 2 VCPU configuration and 5.9% for the 4 VCPU configuration.

The multithreaded benchmark results demonstrate the value of co-scheduling as a SMP VM scheduling algorithm, because the VM's behavior is consistent with that of a physical SMP system. In a physical SMP system, the OS directly controls the scheduling of processes on CPUs. With co-scheduling, a guest OS directly controls the relative scheduling of processes on physical CPUs. This allows the assumptions that are made about system

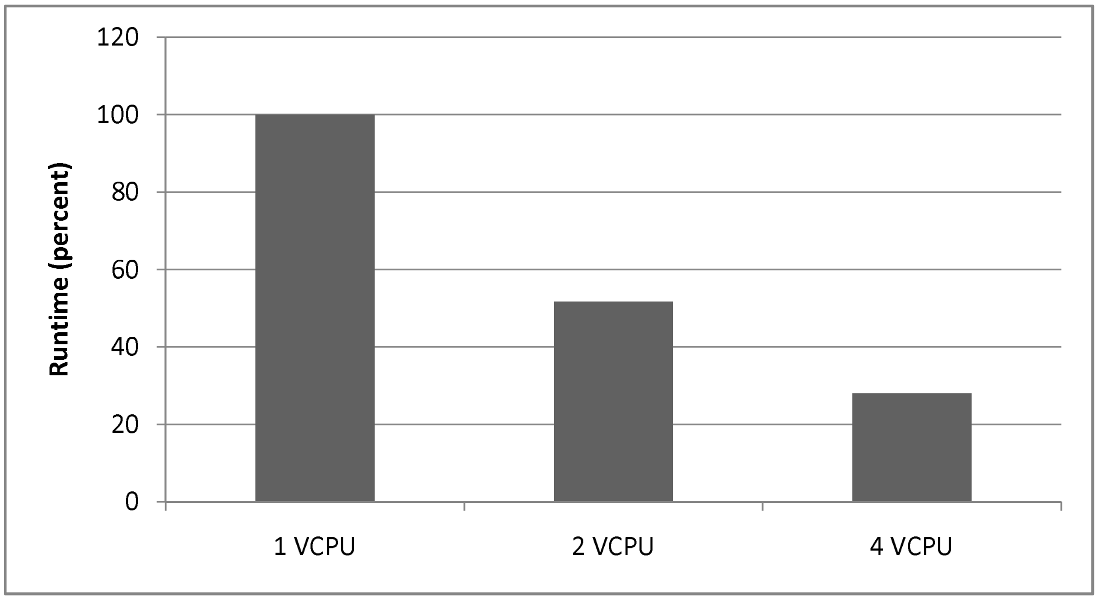

Figure 4.7: Results for 1 VM executing 332.ammp_m with varying numbers of VCPUs

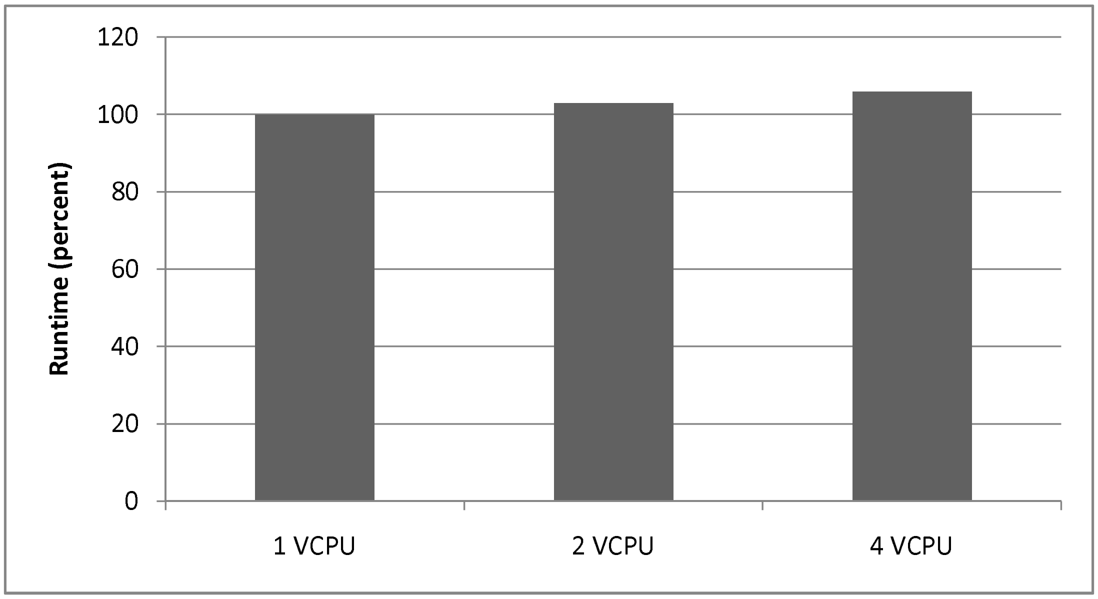

Figure 4.8: Results for 8 VMs executing 332.ammp_m with varying numbers of VCPUs

timing characteristics to remain valid for synchronization between processes in a VM, even if the VM is still unaware of the absolute timing characteristics.

4.4 Xen Benchmark Results

4.4.1 Single Threaded Equal VCPU Allocation Results

The first test conducted was designed to observe overhead incurred by the Xen scheduling algorithm for idle VCPUs. The experiments consisted of 8 VMs, each configured with the same number of VCPUs that were simultaneously executing a single instance of the 445.gobmk benchmark. The experiment was repeated for VMs configured with 1, 2, 4, and 8 VCPUs. Finally, for comparison, a test was executed with a single VM configured with 8 VCPUs executing 8 instances of the 445.gobmk benchmark. The results are shown in figure 4.9.

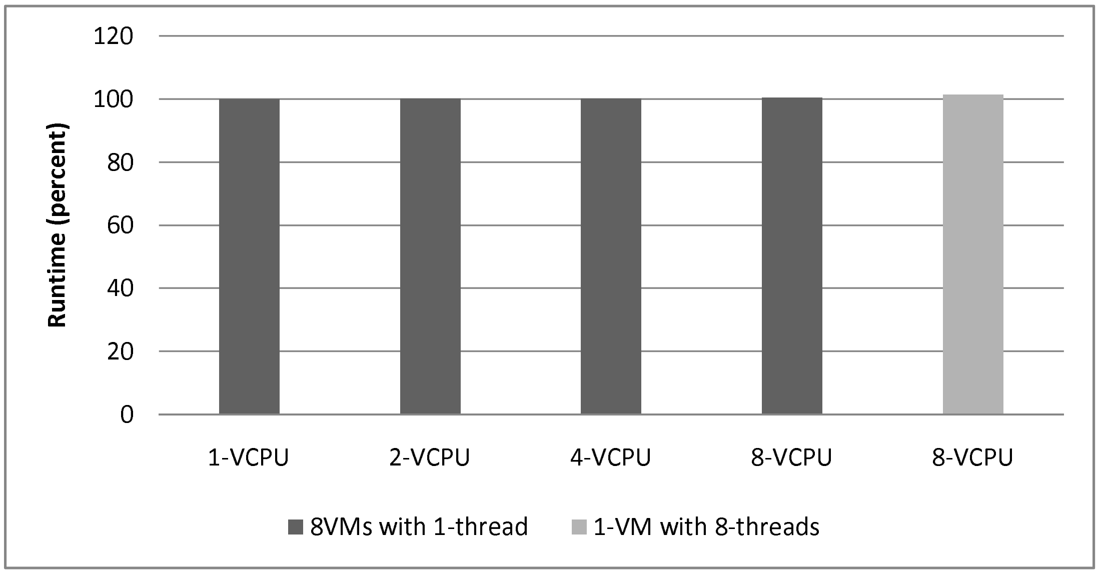

Figure 4.9: Eight VMs simultaneously executing 1 instance of 445.gobmk along with 1 VM executing 8 instances of 445.gobmk. Results normalized to runtime of 1-VM with 1-VCPU executing 1 instance of 445.gobmk.

These results demonstrate that the credit scheduler algorithm used with Xen scales well and has very little overhead. The first set of results shows the performance of the single threaded benchmark as additional VCPUs are added to each VM. There is no change in system performance for benchmark execution, which shows that the scheduling algorithm scales well and that additional VCPUs do not increase system overhead. Also included is the result of executing 8 instances of the single threaded benchmark on a VM configured with 8 VCPUs. The performance here is equivalent to the performance of 8 VMs each executing a single instance of the benchmark. There is less than a 0.5% difference in runtime between any of the VMs executing the single instance of the single threaded benchmarks. The 8 VCPU VM that is executing 8 instances of the benchmark takes 1.5% longer than the one VCPU VM executing one instance of the benchmark. The system performs slightly better when the work is spread among multiple VMs rather than consolidated in a single VM with multiple VCPUs.

The second test shown is similar to the first one because it uses single-threaded benchmarks configured identically on 8 different VMs. However, in this case the performance of the system is measured when CPU resources are overcommitted. The previous benchmark looked at whether idle VCPUs in an SMP VM would cause noticeable overhead when executing single threaded benchmarks. The test performed consisted of 8 VMs each executing 8 instances of the 445.gobmk benchmark. The test was repeated for the case where each VM had 1, 2, 4, and 8 VCPUs. For each of these benchmark tests there are a total of 64 instances of 445.gobmk executing on the host spread over 8 different VMs. The host system where the tests were executed had two quad-core CPUs, resulting in a total of 8 logical processors. For this test there was an 8:1 ratio between processes that could execute and processor resources to execute the processes. The results are shown in figure 4.10. Also included for comparison is a 1-VCPU VM that was executing 8 instances of 445.gobmk.

The results in figure 4.10 look very similar to those in figure 4.9. In this experiment, variation between the 1 VCPU configuration and the 8 VCPU configuration is 0.62%. This shows that the credit scheduler is again doing a good job of evenly partitioning CPU resources among multiple VMs without suffering additional overhead when configured with more VCPUs. The single VM configuration shown at the far right of figure 4.10 is 5%

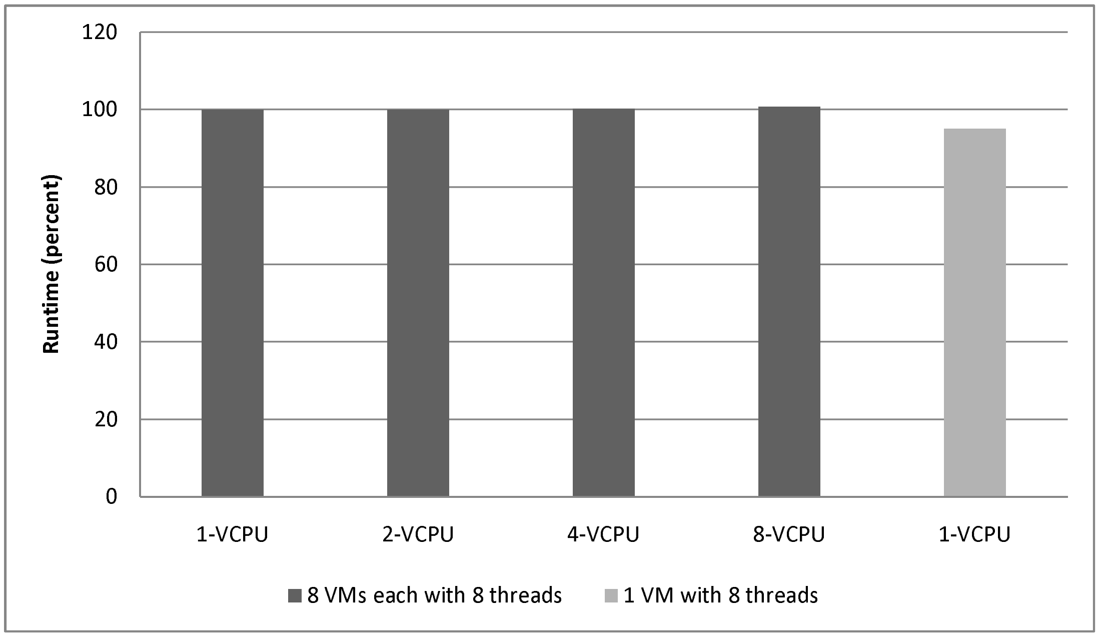

Figure 4.10: Results for 8 VMs simultaneously executing 8 instances of 445.gobmk. Also includes 1 VM executing with 1 VCPU executing 8 instances of 445.gobmk.

faster than the single VM configuration shown on the far left. This performance difference is due to the difference in total workload executing on the host and the fact that the host CPUs have a shared L2 cache. The important thing to observe from these results is that the credit scheduler does not introduce noticeable scheduling overhead when partitioning resources among VMs in an overcommitted system.

Overall, these first two sets of experiments show the advantages of the design decisions made for the SMP scheduling algorithm used by the credit scheduler in Xen. This algorithm has excellent fairness in allocating CPU time among different VMs. In collecting these results, multiple iterations of the benchmarks were run simultaneously on multiple different VMs and the results were averaged. However, the averaging was hardly necessary because the execution time for averaged values was consistently less than 0.1%. The Xen credit scheduler demonstrates excellent fairness and efficiency in allocating CPU resources to SMP VMs executing CPU bound single threaded workloads.

4.4.2 Single Threaded Different VCPU Allocation Results

The next set of results shows the performance for VMs configured with different numbers of VCPUs. The credit scheduler in Xen provides two parameters that can be used to configure the relative performance between VMs. First is the weight value, which determines how to fairly proportion CPU resources between VMs. The second parameter is the cap value, used to limit the maximum amount of CPU resources that a VM can receive. The experiments in this section determine what effect, if any, the number of VCPUs had on fairness of CPU time allocation between VMs.

The first experiment had 8 VMs running simultaneously with each VM executing 8 instances of the 445.gobmk benchmark. This is similar to the scenario used for the last set of results; however, for this experiment the number of VCPUs per VM was not equal. Four of the VMs were assigned 1 VCPU and 4 of the VMs were assigned 8 VCPUs. All VMs were configured with the default weight value of 256, which should have resulted in equal CPU distribution between every VM in the system. One set of results was gathered with the default uncapped configuration, and the other set was gathered with a cap of 100 applied per VM. The cap value is a maximum percentage of CPU resources that a VM is allowed to use. Thus a cap of 100 limits a VM to a maximum of 100% of a single physical CPU. The results for this experiment are shown in figure 4.11. The results were normalized to the 1 VCPU configuration with a cap of 100 applied (third bar from the left in the graph).

These are surprising results, because the number of VCPUs is not supposed to influence the priority of CPU allocation between VMs, according to the design of the credit scheduler. If the scheduling algorithm behaved the way it was documented, we would expect the results for all different VM configurations to be equal — as they were in the first two experiments. For the uncapped configuration the 1 VCPU VMs take 23% longer to complete benchmark execution than the 8 VCPU VMs. In the configuration where CPU resources are capped, the 8 VCPU VMs take 4% longer to execute the benchmark than the 1 VCPU VMs.

The results in figure 4.11 were recorded for a configuration where all VMs had an equal weight. The next experiment looked at the interaction of varied CPU weight configuration

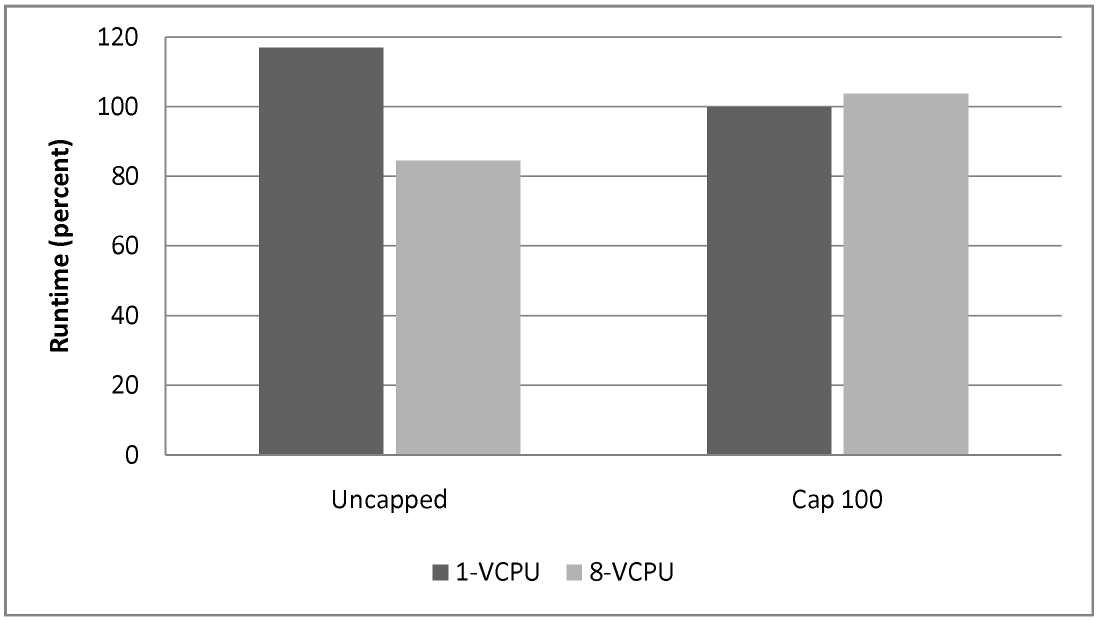

Figure 4.11: Results for VMs executing 8 instances of 445.gobmk with differing VCPU configurations. In each case 4 VMs have 1 VCPU and 4 VMs have 8 VCPUs.

combined with varied numbers of VCPUs. This experiment used only two VMs for each iteration; however, each VM executed 16 instances of the 445.gobmk benchmark. In the first portion of the experiment each VM was configured with 8 VM; however, one VM had a weight of 1024 and the other had a weight of 2048. All results shown in this graph are normalized to the combined execution time of these two VMs. These results are shown in the left hand set of bars in figure 4.12. In the second portion of the experiment one VM was configured with a weight of 1024 and 16 VCPUs and the other VM was configured with a weight of 2048 and 8 VCPUs.

The results shown in figure 4.12 are consistent with those in figure 4.11. Again, the number of VCPUs per VM skews the priority slightly compared to the configured allocations of CPU resources. The left side of the graph, where each VM has 8 VCPUs, has a runtime of 65.8% of the total for the VM with the priority of 1024 and 34.2% for the VM with priority of 2048. The CPU allocation very closely matches the configured weight values. However, in the right side set of values the performance is skewed so that the VM with 16

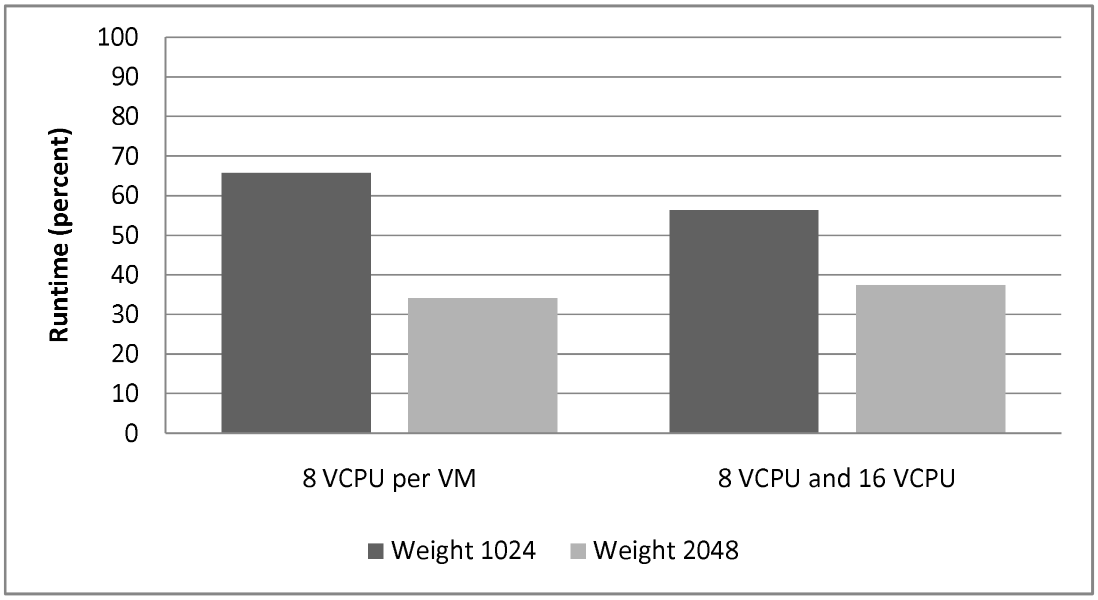

Figure 4.12: Results for VMs executing with differing priority. The left set of values shows two VMs, each with 8 VCPUs. The right set of values shows the VM with weight 1024 with 16 VCPUs and the VM with weight 2048 with 8 VCPUs

VCPUs takes less time than it would otherwise, and the VM with 8 VCPUs takes more time to execute the benchmark than when the VCPUs were allocated evenly. The execution of these benchmarks was not quite as consistent as the earlier results because when one VM finished, the other was able to use all available CPU time. Even with these variations, it is clear that the number of VCPUs per VM influences the priority of CPU allocation in a noticeable way that does not correspond with the weight configuration value.

The final results in this section are shown in figure 4.13. In this experiment, 8 VMs were executing 8 threads of an infinite loop simultaneously to use as much CPU time as possible. Each VM had the default weight of 256 and no cap was configured. Each VM should receive 12.5% of the total host CPU resources, because the CPU time should be divided evenly between all the VMs. However, the actual allocation ranges from 9.3% for the 2 VCPU VM to 13.7% for the 8 VCPU VM.

Overall, the benchmark results recorded in this section show that the number of VCPUs influences the allocation of CPU resources in a way that does not correspond with the

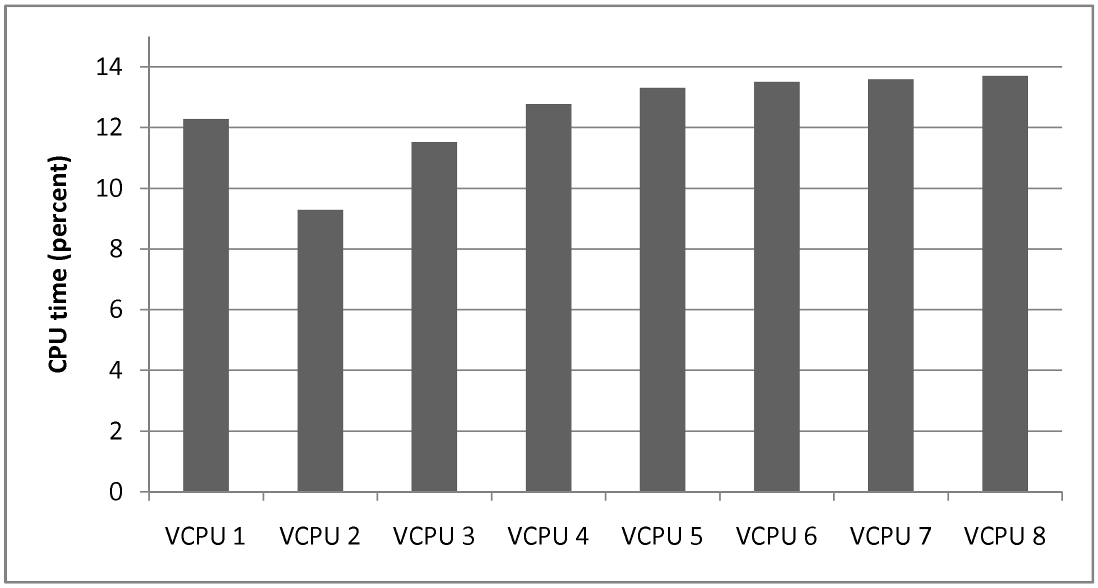

Figure 4.13: Results for 8 VMs executing simultaneously, where each VM is executing 8 threads of an infinite loop to use CPU time. Shows the percentage of host CPU time allocated to VMs based on the number of VCPUs that the VM has. Higher is better.

configured weight values. Prior work on examining the performance of the credit scheduler has shown a lack of fairness for some I/O bound workloads when competing with CPU bound workloads [43] and has also shown flaws with the weight and cap configurations [44].

The prior results focused primarily on testing VMs configured with a single VCPU. The results from this thesis show that the credit scheduler is flawed in how it allocates CPU resources to VMs on an overcommitted host system. I have been in contact with the Xen developers about this problem, and determining a more complete explanation for this behavior is a topic for future work.

4.4.3 Multithreaded benchmarks

The previous sections examined the performance of various configurations of multiple instances of single threaded benchmarks executing on a variety of SMP VM configurations. Multiple largely independent threads executing on the same server are common in workloads such as online transaction processing, where different transactions can be assigned to independent worker threads. However, most parallel programs require a greater level of communication between multiple threads executing in parallel to solve a larger problem. The benchmarks in this section evaluate the performance of the Xen SMP scheduling algorithm for multithreaded workloads that require extensive communication between multiple different threads.

The first experiment executes the 332.ammp_m benchmark on a single VM with varying numbers of VCPUs to establish a performance baseline. The results of this experiment are shown in figure 4.14. Execution time is normalized to the runtime of a VM with a single VCPU. The benchmark shows a nearly linear speedup as the number of VCPUs is increased from 1 to 4, and even the 8 VCPU configuration shows a 6.5 speedup. This demonstrates that the 332.ammp_m benchmark performs well executing on a VM on the test system and has good speedup as the number of VCPUs is increased.

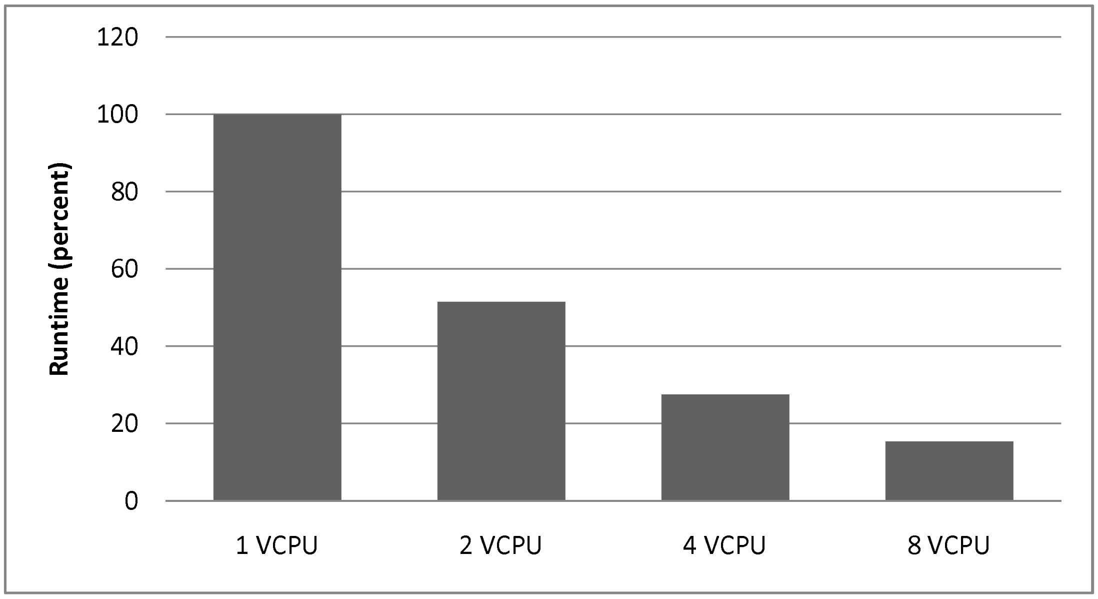

Figure 4.14: Results for a single VM with varying numbers of VCPUs executing the 332.ammp_m benchmark.

The second experiment changes the configuration from a single VM executing the 332.ammp_m benchmark to 8 VMs running the same benchmark with varying numbers of VCPUs. The results in figure 4.15 show an opposite trend from those in figure 4.14. The results in this experiment are again normalized to the 1 VCPU configuration. Here the 2 VCPU configuration is 6% slower, the 4 VCPU configuration is 81% slower, and the 8 VCPU configuration is 1115% percent slower.

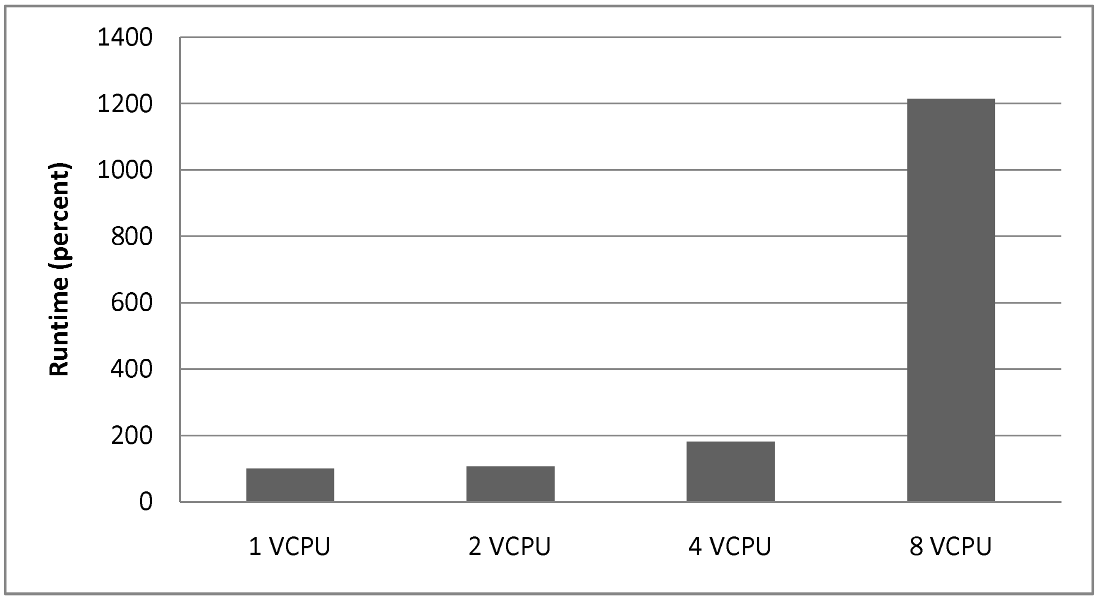

Figure 4.15: Results for a 8 VMs with varying numbers of VCPUs simultaneously executing the 332.ammp_m benchmark.

While it is not expected that adding more VCPUs in an overcommitted system would improve performance, the dramatic slowdown shows a significant disadvantage of the Xen scheduling algorithm's choosing to ignore synchronization requirements between VCPUs. The problem is that OpenMP uses spinlock type synchronization methods to coordinate the multiple threads spawned when executing parallel regions of code. OpenMP attempts to limit the synchronization overhead by determining the number of processors available in the system and then spawning an equal number of threads when executing in parallel regions. This works well in a physical system that is solely dedicated to running an application written using OpenMP, and it also works in a VM if the host system is not CPU bound. However, when there is contention for CPU resources, the performance can drop dramatically, and large amounts of CPU resources can be wasted trying to synchronize VCPUs that are not executing simultaneously.

Table 4.4: ESX and Xen comparison

	ESX	Xen
Strengths	Good multithreaded performance	Supports up to 32 VCPUs per VM
	Optimizations limit co-scheduling overhead	Minimal overhead for independent workloads
Weaknesses	Limited to 4 VCPUs per VM	Performs poorly for synchronized workloads
	Noticeable overhead for idle VCPUs	Number of VCPUs can skew VM priority

4.5 ESX vs. Xen Summary

Both ESX and Xen support SMP virtualization; however, their implementations are very different. Xen uses a fast low overhead scheduling algorithm that allows for up to 32 VCPUs per VM. This algorithm performs very well for independent workloads, idle VCPUs do not cause any noticeable overhead, and running VCPUs are scheduled very efficiently even in an overcommitted system. However, for multithreaded workloads the lack of synchronization can cause problems that significantly degrade system performance. Xen also exhibits a problem where the number of VCPUs per VM skews the allocation of CPU time between VMs in an overcommitted system.

ESX has noticeable performance overhead for idle VCPUs in an SMP VM, with a 12.2% overhead for a 4 VCPU VM. However, ESX performs much better than Xen when executing multithreaded benchmarks. The co-scheduling approach adopted by ESX allows for SMP VMs to execute multithreaded workloads with very low overhead: experiments with the 332.ammp_m benchmark demonstrated a 5.9% overhead.

Although ESX does have a significant performance overhead for idle VCPUs when compared to Xen, it is far better than a naive implementation of a co-scheduling algorithm would be. A naive implementation of co-scheduling that required all VCPUs to execute simultaneously could result in a performance overhead of 100% for each additional VCPU.

This is the type of scenario that previous proposals [9,10] to improve SMP VM scheduling have been compared with. However, VMware's ability to detect and deschedule idle VCPUs, while still using a co-scheduling algorithm, is a significant optimization. Any proposal to improve system performance must take into account these existing optimizations.

Chapter 5: Dynamic VCPU Allocation

Chapter 3 discussed the unique challenges that exist with scheduling VCPUs in an SMP VM. The VMM is often unable to make optimal scheduling decisions because it does not know how the guest OS in a VM is using the VCPUs that it has assigned. If the VCPUs are idle or executing independent threads, then it can be more efficient to schedule VCPUs independently. However, if the VM is executing multithreaded code that requires extensive synchronization, independent VCPU scheduling by the VMM can be very inefficient.

The most common solution to this problem is to have the VMM use co-scheduling whereby all VCPUs assigned to a VM are scheduled simultaneously. The alternative proposals to co-scheduling that were described in Chapter 3 both compared their optimizations against a naive implementation of the co-scheduling algorithm. However, Chapter 4 showed that neither Xen nor VMware ESX uses this naive implementation of co-scheduling. Xen's credit scheduler chooses to ignore the problem of synchronizing VCPUs in an SMP VM. VMware ESX has several optimizations that increase the flexibility and efficiency of their scheduling algorithm.

Chapter 4 showed that Xen's scheduling algorithm exhibits excellent scaling for workloads with independent threads. There was no noticeable overhead in heavily overcommitted workload or in workloads with many idle VCPUs. However, for the highly synchronized multithreaded workload represented by the 332.ammp_m benchmark the Xen scheduling algorithm exhibits very poor performance.

The SMP scheduling algorithm used for VMware ESX has much better performance than Xen for synchronized workloads. But, it too exhibits some overhead as the number of VCPUs per VM scales from 1 to 4. Part of the problem is that parallel workloads such as the 332.ammp_m benchmark do not experience a perfectly linear performance improvement as CPU resources are added. Figure 5.1 shows a comparison of optimal speedup vs. actual speedup for the 332.ammp_m benchmark. The optimal speedup for 8 CPUs would be 8 times; however, the actual speedup is only 6.5 times. The 332.ammp_m benchmark was selected from the OMP2001 benchmark suite because it had a relatively high correlation between added CPUs and benchmark execution speedup.

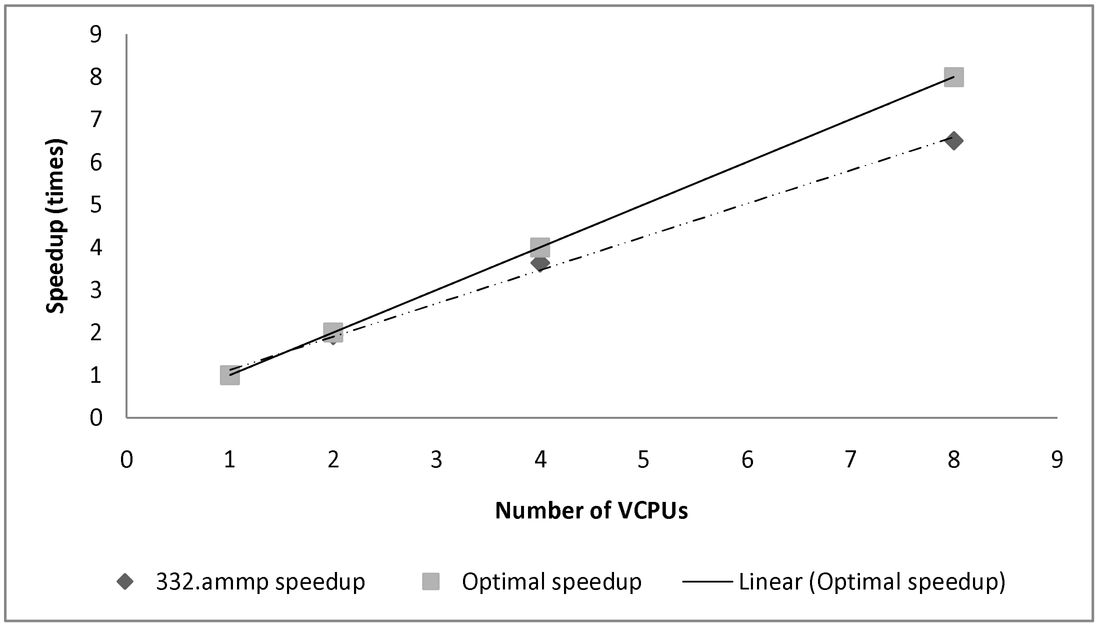

Figure 5.1: Speedup obtained for the 332.ammp_m benchmark with various numbers of VCPUs.

However, even this benchmark does not scale perfectly, and each additional CPU results in a greater percentage of time wasted. VMware ESX currently only supports 1, 2, and 4 VCPU VM configurations, which provides a very limited number of data points for drawing projections about a version of the algorithm that could support greater numbers of VCPUs. However, the 2-VCPU case has a 3% overhead and the 4 VCPU case has a 6% overhead, compared to the 1 VCPU baseline value for the 332.ammp_m benchmark. It is reasonable to predict that if ESX supported more VCPUs it would show a continued increase in performance overhead for VMs with additional VCPUs.

5.1 Proposed Optimization Background

Chapter 3 discussed how changes in computer architecture are forcing the computing industry to find ways to exploit thread-level parallelism rather than rely on performance improvements through increased instruction-level parallelism. Virtualization has emerged as a technology that helps improve the efficiency of multiprocessor utilization, but it does not increase the performance of applications that execute in a virtualized environment. Increasing application performance requires application developers to write programs that can exploit thread-level parallelism, and computer architects to design systems that execute multiple threads simultaneously.

Multithreaded applications have an optimal degree of parallelism that depends on both the structure of the application and hardware the application is executing on and determines how many threads the application should spawn. Often the optimal number of threads is set to be equal to the number of logical processors in the system. The SPEC OMP2001 benchmarks use this technique to automatically determine how many threads to spawn when the benchmark begins executing. For the SPEC OMP2001 benchmarks, the value is determined by checking the number of processors online in the system when the benchmark starts executing. While this works well for a system that is only executing a single benchmark, it is not as efficient on an system executing multiple applications, or on a VM running on an overcommitted host.

5.2 Dynamic VCPU allocation

To improve the performance of multithreaded applications running in a VM, I have explored having the optimal degree of parallelism be determined by considering the workload executing on the entire host system, not just in a single guest VM. Furthermore, I have proposed that the optimal degree of parallelism be determined dynamically and be able to vary as a program is executing. Implementation of this idea proved to be a challenging problem, and one that would require changes at the application level to fully exploit.

Dynamic reconfiguration of the optimal degree of parallelism in a VM requires the following features:

1. The guest OS in a VM must communicate its current degree of parallelism to the VMM.

2. The VMM must be able to communicate to the guest OS what degree of parallelism it can optimally support. It must also be able to change this value dynamically.

3. Applications running in the guest OS must be able to dynamically change their number of threads that are executing based on what the VMM communicates as the optimal degree of parallelism.

In evaluating the potential for this idea, I researched what would be required to implement it using the open-source Xen hypervisor with Linux guests. This provided the possibility of building on an existing system and implementing the idea with a reasonable amount of effort. Each of the requirements is considered in turn:

5.2.1 Communicate current guest OS parallelism to VMM

To communicate the degree of parallelism, the guest OS simply calculates this value and then stores it in a location accessible by the VMM. The maximum degree of parallelism that the guest OS can utilize can be determined by counting the number of runnable and running processes. The load average value that is calculated in Linux provides an exponentially damped weighted average with this information [45]. Xen also provides a mechanism to communicate information from a VM to the VMM using the xenstore. Thus the load average could be used by the VMM to help determine the optimal degree of parallelism for a specific VM.

5.2.2 Communicate optimal parallelism value from VMM to guest

The guest OS executing in a VM is only aware of the processes it is executing and resources it is assigned by the VMM — it does not know about other VMs in the system. Therefore, the guest OS can only communicate its current degree of parallelism to the VMM — it is not able to determine an optimal value for the entire system. However, the VMM collects information about the degree of parallelism from every VM, and it can use this information to calculate an optimal value for each guest OS.

Operating systems are designed to control physical hardware, and their scheduling algorithms work with processes and CPUs. The operating system is not likely to modify its scheduling algorithm to try to take an additional optimal parallelism parameter when it is running in a VMM. Even if it could read this parameter, it is not obvious what it would do with it. Therefore, the way that the VMM communicates this value to the VM is by adjusting the number of VCPUs that are assigned to the guest OS.

Linux has support for adding and removing CPUs from a running system using functionality known as CPU hotplug. In a physical system, this feature can be combined with hardware support to allow CPUs in an multiprocessor system to be replaced on the fly without powering off the machine. Xen supports an extension of this idea, called VCPU hotplug, and allows the number of VCPUs assigned to a VM to be dynamically adjusted without rebooting the VM.

Using the VCPU hotplug feature to adjust the number of VCPUs assigned to a VM effectively adjusts the maximum parallelism that the operating system must manage. Furthermore, applications that check the number of online CPUs to determine how many threads to spawn will see the changed value.

While the first two requirements for a dynamic system are relatively easy to implement, the requirement that applications dynamically reallocate the number of threads they are executing is far harder to fulfill.

The applications in the SPEC OMP2001 benchmark suite are designed to use the number of CPUs that are available in the OS when the application begins execution. The number of threads is not adjusted, even if the number of processors in the system changes during execution. The reason for this is that it is not common for the number of processors to change in a shared memory system while that system is executing. It would be more complicated to design the application logic to support this, and currently there is no reason for developers to write applications that support this capability.

5.2.3 Proposed Algorithm

A full implementation of a dynamic VCPU system is only practical if applications are designed to dynamically adjust their CPU resource utilization based on the current number of VCPUs assigned to a VM. During work on this thesis I did not find a suitable multithreaded application to evaluate my proposal. While I believe that dynamic VCPU allocation remains a viable option, it will require additional work on implementations for OS and virtualization systems, as well as research in programming language, to fully evaluate the proposal. I have begun part of this work by proposing an algorithm to use for dynamic VCPU allocation, and implementing a program in Java to simulate the behavior of this algorithm. The source code for the simulation program is shown in Appendix A. The general outline of the steps in the algorithm are:

1. Calculate total load average of all VMs in the host.
2. Check if the host is overcommitted. This is determined by whether the total load average exceeds the number of PCPUs in the system.
3. Sort the array of VMs by load average in ascending order. This is done so that the VM with the smallest current load average will be processed first.
4. Calculate the target number of VCPUs for each VM. This is done by determining the fraction of remaining host PCPU resources that the VM is eligible to use and adjusting its number of VCPUs accordingly. The list is processed in order, so the VM with the lowest load average has the first opportunity to increase the number of VCPUs it needs. When the VMs with higher load averages are processed, they will be able to increase their number of VCPUs without unfairly using resources from other VMs. For more details on the specific implementation see the function `caclNumCPUs` in Appendix A.

5.3 Summary

Dynamic reallocation of VCPUs between VMs is an interesting idea that may be useful for future programming methodologies as general-purpose applications get better at exploiting thread-level parallelism, yet still need to achieve optimal performance when executing in a VM. Xen has many of the capabilities necessary to build a prototype of such a system; however, today's applications would not be able to take advantage of this type of system. This proposal remains as a possibility for future work that could be explored in conjunction with work examining how to optimize applications to support varying numbers of processors in a system. While this work has focused on varying the number of processors on a VM, these capabilities may also become important in future physical systems as well. Power and thermal characteristics have become a top concern for computer architecture. As the number of cores per chip continues to increase, it is expected that it will be possible to independently turn off some cores while still keeping the processors as a whole running. Such capabilities should help spur further development in OS and application support for CPU hotplug techniques. As this support is added, virtualization systems such as Xen should be able to build on it to extend their capabilities.

Chapter 6: Conclusion and Future Work

Virtualization and multiprocessing are two technologies becoming increasingly important in modern general purpose computing. Until recently, most computers only had a single logical processor, and many operating systems and applications were optimized for uniprocessor systems. Application developers are still adapting to the availability of multiple logical processors in modern computers, and many applications are unable to effectively use these newly available resources.

Virtualization has provided one way to increase the utilization of multiprocessor servers without requiring applications to explicitly support thread-level parallelism. Instead, multiple applications can be combined on a single host, thus increasing the utilization of the host without causing noticeable performance degradation for the VMs running on the host. In fact, in some cases, using multiple independent applications in separate VMs can provide better performance than a single application running on an SMP system. VMware recently published a performance study where SPECweb2005 performed better in a VM than in a large SMP system [46]. Although virtualization has been extremely successful at combining multiple uniprocessor VMs on a multiprocessor host, developing applications that can natively take advantage of multiprocessor resources is an extremely important consideration.

Symmetric multiprocessing virtualization is one way to combine multithreaded applications with the system management advantages offered by virtualization. Support for SMP VMs has become common in many virtualization systems, including two of the leading virtualization solutions: VMware ESX and Xen. Implementing a high performance scheduling algorithm for SMP VMs is difficult, because virtualization changes some of the timing characteristics of VMs in ways that the guest OS might not expect. This can cause problems with some synchronization routines running in a VM.

Xen's scheduling algorithm chooses to ignore synchronization concerns when scheduling the VCPUs associated with an SMP VM. This provides high performance and low overhead for single threaded processes; however, it results in a significant performance degradation for highly synchronized workloads such as those associated with OpenMP. The credit scheduler algorithm used by Xen also performs unpredictably in its allocation of CPU resources in an overcommitted system.

ESX's scheduling algorithm uses a more conservative approach, called co-scheduling, in which all VCPUs associated with a VM are scheduled concurrently. However, ESX has some important optimizations to limit the amount of wasted CPU time and to increase VCPU scheduling flexibility. The result is that for single threaded workloads, ESX shows a very reasonable performance overhead, and the performance also scales well for multithreaded workloads.

Existing proposals for optimizing SMP VM scheduling have been compared to naive co-scheduling implementation, rather than to the highly optimized co-scheduling approach adopted by ESX, or the choice to ignore synchronization concerns adopted by Xen. This thesis provides a quantitative analysis of these two existing systems, and shows that in many cases they are already providing excellent performance.

Even in the best case, however, a parallel application is unlikely to experience a speedup equal to the number of CPUs. This is because only part of the application is likely to be suitable for parallelizing, and the sequential portion will limit the total speedup. This limitation is often formulated as Amdahl's law. In this thesis I have proposed a way to optimize the performance of SMP VMs through adapting the number of VCPUs assigned to a VM based on the total host system utilization.

As future work, I plan to implement the algorithm that I have proposed for dynamic VCPU allocation in Xen. My initial implementation is useful as a simulation, but more extensive testing will require an actual implementation. I also plan to continue researching options for available benchmarks to provide a quantitative analysis of the dynamic VCPU scheduling proposal. Finally, I have been in contact with some of the Xen developers to try to determine why total host CPU allocation is influenced by the number of VCPUs assigned to a VM. I plan to continue to study this problem to try to determine a more complete explanation for this behavior.

Appendix A: VCPU Allocation Source Code

```java
package thesis;

import java.util.Arrays;
import java.util.Scanner;

/**
 *
 * @author Gabriel Southern
 */
class VM implements Comparable<VM> {

    static final int MAX_VCPUS = 32;
    private int numVCPUs;
    private double loadAvg;
    private int id;

    public VM(int numVCPUs, double loadAvg, int id) {
        this.numVCPUs = numVCPUs;
        this.loadAvg = loadAvg;
        this.id = id;
    }

    public int getId() {
        return id;
    }

    public void removeVCPU() {
        if (numVCPUs > 0) {
            numVCPUs--;
        }
    }

    public void addVCPU() {
        if (numVCPUs < MAX_VCPUS) {
            numVCPUs++;
        }
    }

    public double getLoadAvg() {
        return loadAvg;
    }

    public void setLoadAvg(double la) {
        if (la >= 0) {
            this.loadAvg = la;
        }
    }

    public int getNumVCPUs() {
        return numVCPUs;
    }

    public int compareTo(VM o) {
        if (this.loadAvg < o.loadAvg) {
            return -1;
        }
        if (this.loadAvg == o.loadAvg) {
            return 0;
        }
        if (this.loadAvg > o.loadAvg) {
            return 1;
        } else {
            return -1;
        }
    }
}

public class DynVCPU {

    static private int numPCPUs;
    static private int numVMs;

    public static void main(String[] args) {
        Scanner userInput = new Scanner(System.in);

        System.out.println("Dynamic VCPU allocation simulator");
        System.out.print("Enter number of PCPUs in host: ");
        numPCPUs = userInput.nextInt();
        System.out.print("Enter number of VMs on host: ");
        numVMs = userInput.nextInt();
        System.out.println("Number of VMs is: " + numVMs);

        VM[] vms = new VM[numVMs];
        for (int i = 0; i < numVMs; i++) {
            System.out.println("Enter number of VCPUs for VM"
                    + i + ": ");
            int numvcpus = userInput.nextInt();
            System.out.println("Enter initial load average for VM"
                    + i + ": ");
            double loadavg = userInput.nextDouble();
            vms[i] = new VM(numvcpus, loadavg, i);
        }

        while (true) {
            calcNumCPUs(vms);
            System.out.println("\nNew VCPU allocation should be:");
            for (VM v : vms) {
                System.out.println("VM " + v.getId() + " is: " +
                    v.getNumVCPUs() + " current load average is: "
                    + v.getLoadAvg());
            }
            System.out.println("\nUpdate system config\n");

            for (VM v : vms) {
                System.out.println("Enter loadavg for VM"
                        + v.getId() + ": ");
                v.setLoadAvg(userInput.nextDouble());
            }

        }

    }

    /**
     * Updates the number of VCPUs that are assigned to a VM
     * using an algorithm that tries to match the number of VCPUs
     * to the load average
     * @param vmList
     */
    static void calcNumCPUs(VM[] vmList) {

        double hostLoadAvg = 0;
        for (VM v : vmList) {
            hostLoadAvg += v.getLoadAvg();
        }

        boolean overcommitted;
        if (hostLoadAvg >= DynVCPU.numPCPUs) {
            overcommitted = true;
        } else {
            overcommitted = false;
        }

        Arrays.sort(vmList);

        int availableCPUs = DynVCPU.numPCPUs;
        int remainingVMs = DynVCPU.numVMs;
        double currentLoadAvg = hostLoadAvg;

        for (VM v : vmList) {
            // Calcuate maximum allowed VCPUs per VM
            int maxVcpus = (int)Math.ceil(Math.max(1.0,
                 (availableCPUs / Math.max(currentLoadAvg, 1.0)))
                  * Math.max(1.0, (DynVCPU.numPCPUs/remainingVMs)));

            // Update VM VCPU allocation and host CPU stats
            if (v.getNumVCPUs() > maxVcpus) {
                v.removeVCPU();
                availableCPUs -= Math.min(maxVcpus, v.getLoadAvg());
            } else if (v.getNumVCPUs() == maxVcpus) {
                availableCPUs -= Math.min(maxVcpus, v.getLoadAvg());
            } else if (v.getLoadAvg() > v.getNumVCPUs()) {
                v.addVCPU();
                availableCPUs -= v.getLoadAvg();
            } else if (v.getLoadAvg() == v.getNumVCPUs()
                    && !overcommitted) {
                v.addVCPU();
                availableCPUs -= v.getLoadAvg();
            } else {
                availableCPUs -= v.getLoadAvg();
            }

            // Update host load average for next VM
            currentLoadAvg -= v.getLoadAvg();
            --remainingVMs;
        }
    }
}
```

Bibliography

Bibliography

[1] G. J. Popek and R. P. Goldberg, "Formal requirements for virtualizable third generation architectures," *Communications of the ACM*, vol. 17, no. 7, pp. 412–421, Jul. 1974.

[2] M. Rosenblum and T. Garfinkel, "Virtual machine monitors: Current technology and future trends," *IEEE Computer*, vol. 38, pp. 39–47, May 2005.

[3] J. S. Robin and C. E. Irvine, "Analysis of the Intel Pentium's ability to support a secure virtual machine monitor," in *Proceedings of the Ninth USENIX Security Symposium*, Denver, Colorado, Aug. 2000.

[4] VMware ESX 3.5 product datasheet. VMware. Palo Alto, CA. [Online]. Available: http://www.vmware.com/files/pdf/esx_datasheet.pdf

[5] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," in *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, Bolton Landing, New York, Oct. 2003, pp. 164–177.

[6] "Best practices using VMware virtual SMP," White Paper, VMware, 2005. [Online]. Available: http://www.vmware.com/pdf/vsmp_best_practices.pdf

[7] (2007, Jan.) Performance tuning best practices for esx server 3. VMware. [Online]. Available: http://www.vmware.com/pdf/vi_performance_tuning.pdf

[8] "Xenserver virtual machine installation guide," User manual, Citrix, 2007. [Online]. Available: http://support.citrix.com/article/CTX115642

[9] V. Uhlig, J. LeVasseur, E. Skoglund, and U. Dannowski, "Towards scalable multiprocessor virtual machines," in *Virtual Machine Research and Technology Symposium*, San Jose, California, May 2004, pp. 43–56.

[10] P. M. Wells, K. Chakraborty, and G. S. Sohi, "Hardware support for spin management in overcommitted virtual machines," in *Proceedings of the 15th International Conference on Parallel Architecture and Compilation Techniques (15th PACT'06)*, Seattle, Washington, USA, Sep. 2006, pp. 124–133.

[11] A. Silberschatz, P. Galvin, and G. Gange, *Operating System Concepts*. Hoboken, New Jersey: John Wiley & Sons, Inc., 2005.

[12] A. Leon-Garcia and I. Widjaja, *Communication Networks*. New York, NY: McGraw-Hill, 2004.

[13] E. Bugnion, S. Devine, and M. Rosenblum, "Disco: Running commodity operating systems on scalable multiprocessors," in *Proceedings of the 16th Symposium on Operating Systems Principles (SOSP-97)*, Saint-Malo, France, Oct. 1997, pp. 143–156.

[14] "The future of virtualization," Information Week Magazine, CMP, Aug. 20, 2007.

[15] J. Sugerman, G. Venkitachalam, and B.-H. Lim, "Virtualizing I/O devices on VMware workstation's hosted virtual machine monitor," in *USENIX Annual Technical Conference, General Track*, Boston, Massachusetts, Jun. 2001, pp. 1–14.

[16] "VMware vs. microsoft," Information Week Magazine, CMP, Oct. 15, 2007.

[17] "Microsofts hypervisor technology gives customers combined benefits of windows server 2008 and virtualization," Press Release, Microsoft, Jun. 2008. [Online]. Available: http://www.microsoft.com/presspass/features/2008/jun08/06-26hyperv.mspx

[18] G. M. Amdahl, G. A. Blaauw, and F. P. Brooks, Jr., "Architecture of the IBM System/360," *IBM Journal of Research and Development*, vol. 44, no. 1/2, pp. 21–36, Jan./Mar. 2000, special issue: reprints on Evolution of information technology 1957–1999. [Online]. Available: http://www.research.ibm.com/journal/rd/441/amdahl.pdf

[19] R. J. Creasy, "The origin of the VM/370 time-sharing system," *IBM Journal of Research and Development*, vol. 25, no. 5, pp. 483–490, 1981.

[20] "EMC to acquire VMware, advancing convergence of server and storage virtualization," Press Release, VMware, Dec. 2003. [Online]. Available: http://www.vmware.com/company/news/releases/emc.html

[21] J. E. Smith and R. Nair, *Virtual Machines: Versatile Platforms for Systems and Processes.* San Francisco, California: Morgan Kaufmann Publishers, 2005.

[22] K. Adams and O. Agesen, "A comparison of software and hardware techniques for x86 virtualization," in *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2006*, San Jose, California, Oct. 2006, pp. 2–13.

[23] C. A. Waldspurger, "Memory resource management in VMware ESX server," in *Proceedings of the 5th ACM Symposium on Operating System Design and Implementation (OSDI-02)*, Boston, Massachusetts, Dec. 2002, pp. 181–194.

[24] "Citrix to enter server and desktop virtualization markets with acquisition of xensource," Press Release, Citrix, Aug. 2007. [Online]. Available: http://www.citrix.com/English/NE/news/news.asp?newsID=680808

[25] R. Uhlig, G. Neiger, D. Rodgers, A. L. Santoni, F. C. M. Martins, A. V. Anderson, S. M. Bennett, A. Kägi, F. H. Leung, and L. Smith, "Intel virtualization technology," *IEEE Computer*, vol. 38, no. 5, pp. 48–56, 2005. [Online]. Available: http://doi.ieeecomputersociety.org/10.1109/MC.2005.163

[26] Y. Dong, S. Li, A. Mallick, J. Nakajim, K. Tian, X. Xu, F. Yang, and W. Yu, “Extending Xen with Intel virtualization technology,” *Intel Technology Journal*, vol. 10, no. 3, pp. 193–203, Aug. 2006. [Online]. Available: http://developer.intel.com/technology/itj/2006/v10i3/3-xen/1-abstract.htm

[27] J. von Neumann, “First draft of a report on the EDVAC,” Moore School of Electrical Engineering, University of Pennsylvania, Tech. Rep., 1945. [Online]. Available: http://qss.stanford.edu/~godfrey/vonNeumann/vnedvac.pdf

[28] M. Flynn, “Some computer organizations and their effectiveness,” *IEEE Transactions on Computers*, vol. C-21, pp. 948–960, Sep. 1972.

[29] J. P. Shen and M. H. Lipasti, *Modern Processor Design.* New York, NY: McGraw-Hill, 2005.

[30] J. Wawrzynek, D. Patterson, M. Oskin, S.-L. Lu, C. Kozyrakis, J. C. Hoe, D. Chiou, and K. Asanović, “RAMP: Research accelerator for multiple processors,” *IEEE Micro*, vol. 27, pp. 46–57, Mar./Apr. 2007.

[31] D. Koufaty and D. T. Marr, “Hyperthreading technology in the netburst microarchitecture,” *IEEE Micro*, vol. 23, no. 2, pp. 56–65, 2003.

[32] H. McGhan, “Niagara 2 opens the floodgates,” Microprocessor Report, Reed Electronics Group, Nov. 2006. [Online]. Available: http://www.sun.com/processors/niagara/M45_MPFNiagara2_reprint.pdf

[33] (2008, Jun.) June 2008 top 500 supercomputing sites. Website: top500.org. [Online]. Available: http://www.top500.org/lists/2008/06

[34] P. Conway and B. Hughes, “The AMD opteron northbridge architecture,” *IEEE Micro*, vol. 27, no. 2, pp. 10–21, 2007.

[35] V. Uhlig, “The mechanics of in-kernel synchronization for a scalable microkernel,” *SIGOPS Oper. Syst. Rev.*, vol. 41, no. 4, pp. 49–58, 2007.

[36] T. Li, A. R. Lebeck, and D. J. Sorin, “Spin detection hardware for improved management of multithreaded systems,” *IEEE Transactions on Parallel and Distributed Systems*, vol. PDS-17, no. 6, pp. 508–521, Jun. 2006.

[37] D. Chisnall, *The Definitive Guide to the Xen Hypervisor.* Boston, Massachusetts: Prentice Hall, 2007.

[38] (2007, Nov.) Resource management guide: ESX server 3.5, ESX server 3i version 3.5, VirtualCenter 2.5. VMware. Revision 20071129. [Online]. Available: http://www.vmware.com/pdf/vi3_35/esx_3/r35/vi3_35_25_resource_mgmt.pdf

[39] (2008, May) Co-scheduling smp vms in VMware ESX server. VMware. Version 3. [Online]. Available: http://communities.vmware.com/docs/DOC-4960.pdf

[40] (2008, Jul.) The SPEC organization. Website: www.spec.org. [Online]. Available: http://www.spec.org/spec/

[41] V. Aslot and R. Eigenmann, "Performance characteristics of the spec omp2001 benchmarks," *SIGARCH Comput. Archit. News*, vol. 29, no. 5, pp. 31–40, 2001.

[42] (2008, Jul.) Xen users' manual. Xen.org. [Online]. Available: http://bits.xensource.com/Xen/docs/user.pdf

[43] D. Ongaro, A. L. Cox, and S. Rixner, "Scheduling i/o in virtual machine monitors," in *VEE '08: Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, Seattle, WA, USA, Mar. 2008, pp. 1–10.

[44] L. Cherkasova, D. Gupta, and A. Vahdat, "Comparison of the three cpu schedulers in xen," *SIGMETRICS Perform. Eval. Rev.*, vol. 35, no. 2, pp. 42–51, 2007.

[45] R. Walker. (2006, Dec.) Examining load average. Linux Journal. [Online]. Available: http://www.linuxjournal.com/article/9001

[46] (2008, May) Consolidating web applications using VMware infrastructure. Performance Study. VMware. [Online]. Available: http://www.vmware.com/files/pdf/consolidating_webapps_vi3_wp.pdf

Curriculum Vitae

Gabriel Southern received his Bachelor of Science in Computer Engineering in 2002 from the University of Virginia. He then was commissioned as an officer in the U.S. Army, serving on active duty for four years and rising to the rank of captain. In 2006 he enrolled in the Electrical and Computer Engineering Department at George Mason University, earning his Master of Science in Computer Engineering in 2008. At graduation he was recognized for outstanding academic achievement, and he was the recipient of the 2008 Outstanding Graduate Award given by the Volgenau School of Information Technology and Engineering. While a student, he was employed as an information technology integration engineer at Computer Systems Center Inc. (CSCI). He will be pursing his Ph.D. in computer engineering at the University of California, Santa Cruz.